Angewandte Komplexitätstheorie

Fachhochschule Nordostniedersachsen

Fachbereich Wirtschaft

Prof. Dr. Ulrich Hoffmann

November 2000

Inhaltsverzeichnis

1 Einleitung

1.1 Angewandte Komplexitätstheorie

Die **Komplexitätstheorie** untersucht den Aufwand, den die Ausführung von Algorithmen erfordert, und zwar sowohl den zeitlichen Aufwand als auch den Speicherplatzbedarf bei der Ausführung der Algorithmen in Rechenanlagen. Seit Beginn der 1960‘er Jahre hat sie eine stürmische Entwicklung erlebt. Die zentrale Stelle dieser Theorie bildet das **P**-**NP**-Problem, das bis heute ungelöst ist. Die Beschäftigung mit diesem Problem hat eine Reihe bedeutender Erkenntnisse darüber geliefert, mit welchem Aufwand Probleme und Problemklassen lösbar sind bzw. nicht gelöst werden können, und hat letztlich immer wieder zu neuen Designtechniken für Algorithmen geführt. Zu diesen neuen Erkenntnissen gehört das sogenannte PCP-Theorem, das die Bedeutung probabilistischer Ansätze in der Algorithmentheorie hervorhebt. Es wird als das spektakulärste Ergebnis der Theoretischen Informatik in den letzten 10 Jahren angesehen.

Einen breiten Raum innerhalb der Komplexitätstheorie nimmt die Untersuchung von mehr oder weniger abstrakten Problemklassen ein. Diese bauen auf den Möglichkeiten universeller Turingmaschinen auf und liefern Erkenntnisse über die Struktur ganzer Problemklassen. Der vorliegende Text konzentriert sich mehr auf die Darstellung anwendungsbezogener Aspekte der Komplexitätstheorie: daher die Bezeichnung **Angewandte Komplexitätstheorie**. Auch dieser Bereich der Komplexitätstheorie hat inzwischen einen Umfang angenommen, der innerhalb einer einsemestrigen Vorlesung nicht mehr umfassend dargestellt werden kann, so daß eine (subjektive) Themenauswahl getroffen werden mußte. Ziel ist es, wichtige Fragestellungen auch anhand konkreter Beispiele zu erläutern. Dabei wird ein Schwerpunkt auf Optimierungsproblemen und deren Approximierbarkeit liegen. Natürlich werden grundlegende Konzepte behandelt, sofern sie zum Verständnis der Fragen der Lösbarkeit notwendig sind.

Zentrale Fragestellungen der Angewandten Komplexitätstheorie sind u.a.

- das Auffinden von Rechenverfahren, die ein vorgegebenes Problem, d.h. eine vorgegebene Aufgabenstellung, lösen
- die Definition dazu geeigneter Datenstrukturen
- die Festlegung von „Gütekriterien“ für Algorithmen
- das Laufzeitverhalten und dem Speicherplatzverbrauch bei der Lösung eines Problems
- der Vergleich von Algorithmen, die ein bestimmtes Problem lösen

1.2 Probleme und Algorithmen

Ein **Problem** ist eine zu beantwortende Fragestellung, die von Problemparametern (Variablenwerten, Eingaben usw.) abhängt, deren genaue Werte in der Problembeschreibung zunächst unspezifiert sind. Ein Problem wird beschrieben durch:

1. eine allgemeine Beschreibung aller Parameter, von der die Problemlösung abhängt; diese Beschreibung spezifiziert die (Problem-) **Instanz**

2. die Eigenschaften, die die Antwort, d.h. die Problemlösung, haben soll.

Eine spezielle Problemstellung erhält man durch Konkretisierung einer Probleminstanz, d.h. durch die Angabe spezieller Parameterwerte in der Problembeschreibung.

Im folgenden werden einige grundlegende **Problemtypen** unterschieden und zunächst an Beispielen erläutert.

Problem des Handlungsreisenden als Optimierungsproblem
(minimum traveling salesperson problem)

Instanz: $x = (C, d)$
$C = \{c_1, ..., c_n\}$ ist eine endliche Menge und $d : C \times C \to \mathbf{N}$ eine Funktion, die je zwei $c_i \in C$ und $c_j \in C$ eine nichtnegative ganze Zahl $d(c_i, c_j)$ zuordnet.

Die Menge *C* kann beispielsweise als eine Menge von Orten und die Wert $d(c_i, c_j)$ können als Abstand zwischen c_i und c_j interpretiert werden.

Lösung: Eine Permutation (Anordnung) $p : [1:n] \to [1:n]$, d.h. (implizit) eine Anordnung $\langle c_{p(1)}, ..., c_{p(n)} \rangle$ der Werte in *C*, die den Wert

$$\sum_{i=1}^{n} d\left(c_{p(i)}, c_{p(i+1)}\right) + d\left(c_{p(n)}, c_{p(1)}\right)$$

minimiert (unter allen möglichen Permutationen von $[1:n]$).
Dieser Ausdruck gibt die Länge einer „kürzesten Tour“ an, die in $c_{p(1)}$ startet, nacheinander alle Orte einmal besucht und direkt vom letzten Ort $c_{p(n)}$ nach $c_{p(1)}$ zurückkehrt.

Ein Beispiel einer Instanz ist die Menge $C = \{c_1, ..., c_4\}$ mit den Werten $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, $d(c_3, c_4) = 3$ und $d(c_i, c_j) = d(c_j, c_i)$.

Die Permutation $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 4 & 3 \end{pmatrix}$, d.h. die Anordnung $\langle c_1, c_2, c_4, c_3 \rangle$ der Werte in *C* ist eine (optimale) Lösung mit Wert 27.

Eine verallgemeinerte Formulierung des Problems des Handlungsreisenden bezogen auf endliche Graphen lautet:

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Eine **Tour** durch *G*, d.h. eine geschlossene Kantenfolge

$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ mit $(v_{i_j}, v_{i_{j+1}}) \in E$ für $j = 1, ..., n-1$,

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht. Man kann o.B.d.A. $v_{i_1} = v_1$ setzen.

Die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist dabei die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

$$\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1}).$$

Das Problem des Handlungsreisenden findet vielerlei Anwendungen in den Bereichen

- Transportoptimierung:
  Ermittlung einer kostenminimalen Tour, die im Depot beginnt, $n-1$ Kunden erreicht und im Depot endet.

- Fließbandproduktion:
  Ein Roboterarm soll Schrauben an einem am Fließband produzierten Werkstück festdrehen. Der Arm startet in einer Ausgangsposition (über einer Schraube), bewegt sich dann von einer zur nächsten Schraube (insgesamt *n* Schrauben) und kehrt in die Ausgangsposition zurück.

- Produktionsumstellung:
  Eine Produktionsstätte stellt verschiedene Artikel mit denselben Maschinen her. Der Herstellungsprozeß verläuft in Zyklen. Pro Zyklus werden *n* unterschiedliche Artikel produziert. Die Änderungskosten von der Produktion des Artikels v_i auf die des Artikels v_j betragen $w((v_i, v_j))$ (Geldeinheiten). Gesucht wird eine kostenminimale Produktionsfolge. Das Durchlaufen der Kante (v_i, v_j) entspricht dabei der Umstellung von Artikel v_i auf Artikel v_j. Gesucht ist eine Tour (zum Ausgangspunkt zurück), weil die Kosten des nächsten, hier des ersten, Zyklusstarts mit einbezogen werden müssen.

Problem des Handlungsreisenden auf Graphen als Berechnungsproblem

Instanz: $G = (V, E, w)$
$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Der Wert $\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$ einer kostenminimalen Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch *G*.

Problem des Handlungsreisenden auf Graphen als Entscheidungsproblem

Instanz: $G = (V, E, w)$ und $K \in \mathbf{R}_{\geq 0}$
$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Die Antwort auf die Frage:
Gibt es eine kostenminimale Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch *G*, deren Wert $\leq K$ ist?

Die Instanz *x* eines Problems Π ist eine endliche Zeichenkette über einem endlichen Alphabet Σ_Π, das dazu geeignet ist, derartige Problemstellungen zu formulieren, d.h. $x \in \Sigma_\Pi^*$ [1].

Es werden folgende **Problemtypen** unterschieden:

Entscheidungsproblem Π:

Instanz: $x \in \Sigma_\Pi^*$
und Spezifikation einer Eigenschaft, die einer Auswahl von Elementen aus Σ^* zukommt, d.h. die Spezifikation einer Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit
$L_\Pi = \{u \in \Sigma^* \mid u \text{ hat die beschriebene Eigenschaft}\}$

Lösung: Entscheidung „ja“, falls $x \in L_\Pi$ ist,
Entscheidung „nein“, falls $x \notin L_\Pi$ ist.

Bemerkung: In konkreten Beispielen für Entscheidungsprobleme wird gelegentlich die Problemspezifikation nicht in dieser strengen formalen Weise durchgeführt. Insbesondere wird die Spezifikation der die Menge $L_\Pi \subseteq \Sigma_\Pi^*$ beschreibenden Eigenschaft direkt bei der Lösung angegeben.

Berechnungsproblem Π:

Instanz: $x \in \Sigma_\Pi^*$
und die Beschreibung einer Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$.

Lösung: Berechnung des Werts $f(x)$.

[1] Üblicherweise bezeichnet A^* die Menge aller Zeichenketten endlicher Länge, die man mit Hilfe der Zeichen eines endlichen Alphabets *A* bilden kann; A^* heißt auch **Menge der Wörter über dem (endlichen) Alphabet** *A*.

Optimierungsproblem Π:

Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ **eine Menge zulässiger Lösungen** zuordnet
3. Spezifikation einer **Zielfunktion** m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den **Wert einer zulässigen Lösung**, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$) bzw. $m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

In dieser (formalen) Terminologie wird das Handlungsreisenden-Minimierungsproblem wie folgt formuliert:

Instanz:
1. $G = (V, E, w)$
   $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht
2. $\mathrm{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist eine Tour durch $G\}$;
   eine Tour durch G ist eine geschlossene Kantenfolge, in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt
3. für $T \in \mathrm{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$
4. $goal = \min$

Lösung: Eine Tour $T^* \in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

Ein **Algorithmus** ist eine Verfahrensvorschrift (Prozedur, Berechnungsvorschrift), die aus einer endlichen Menge eindeutiger Regeln besteht, die eine endliche Aufeinanderfolge von Operationen spezifiziert, so daß eine Lösung zu einem Problem bzw. einer spezifischen Klasse von Problemen daraus erzielt wird.

Konkret kann man sich einen Algorithmus als ein Computerprogramm vorstellen, das in einer Programmiersprache formuliert ist.

Typische **Fragestellungen** bei einem gegebenen Algorithmus für eine Problemlösung sind:

- Hält der Algorithmus immer bei einer gültigen Eingabe nach endlich vielen Schritten an?

- Berechnet der Algorithmus bei einer gültigen Eingabe eine korrekte Antwort?
  Die positiver Beantwortung beider Fragen erfordert einen mathematischen **Korrektheitsbeweis** des Algorithmus. Bei positiver Beantwortung nur der zweiten Frage spricht man von **partieller Korrektheit**. Für die partielle Korrektheit ist lediglich nachzuweisen, daß der Algorithmus bei einer gültigen Eingabe, bei der er nach endlich vielen Schritten anhält, ein korrektes Ergebnis liefert.

- Wieviele Schritte benötigt der Algorithmus bei einer gültigen Eingabe **höchstens (worst case analysis)** bzw. **im Mittel (average case analysis)**, d.h. welche **(Zeit-) Komplexität** hat er im schlechtesten Fall bzw. im Mittel? Dabei ist es natürlich besonders interessant nachzuweisen, daß die Komplexität des Algorithmus von der jeweiligen Formulierungsgrundlage (Programmiersprache, Maschinenmodell) weitgehend unabhängig ist.

  Entsprechend kann man nach dem benötigten **Speicherplatzaufwand (Platzkomplexität)** eines Algorithmus fragen.

  Die Beantwortung dieser Fragen gibt obere Komplexitätsschranken (Garantie für das Laufzeitverhalten bzw. den Speicherplatzverbrauch) an.

- Gibt es zu einer Problemlösung eventuell ein „noch besseres“ Verfahren (mit weniger Rechenschritten, weniger Speicherplatzaufwand)? Wieviele Schritte wird jeder Algorithmus mindestens durchführen, der das vorgelegte Problem löst?

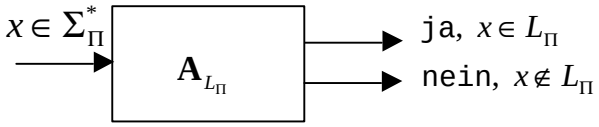
  Die Beantwortung dieser Frage liefert untere Komplexitätsschranken.

Entsprechend der verschiedenen Problemtypen (Entscheidungsproblem, Berechnungsproblem, insbesondere Optimierungsproblem) gibt es auch unterschiedliche **Typen von Algorithmen**:

Ein **Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: ja, falls $x \in L_\Pi$ gilt
nein, falls $x \notin L_\Pi$ gilt.



Mit $\mathbf{A}_{L_\Pi}(x)$, $\mathbf{A}_{L_\Pi}(x) \in \{\texttt{ja, nein}\}$, wird die Entscheidung von $\mathbf{A}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ bezeichnet. Zu beachten ist, daß $\mathbf{A}_{L_\Pi}$ bei jeder Eingabe $x \in \Sigma_\Pi^*$ hält.

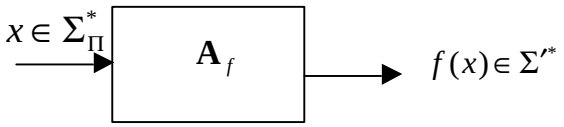
Die folgenden Sprechweisen werden synonym verwendet:

- $\mathbf{A}_{L_\Pi}$ löst das Entscheidungsproblem Π.
- $\mathbf{A}_{L_\Pi}$ **akzeptiert** die Menge L_Π (mit $L_\Pi \subseteq \Sigma_\Pi^*$).
- $\mathbf{A}_{L_\Pi}$ **erkennt** die Menge L_Π (mit $L_\Pi \subseteq \Sigma_\Pi^*$).
- $\mathbf{A}_{L_\Pi}$ **entscheidet** die Menge L_Π (mit $L_\Pi \subseteq \Sigma_\Pi^*$).

Ein **Algorithmus $\mathbf{A}_f$ für ein Berechnungsproblem** Π (Berechnung einer Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$) hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: $f(x) \in \Sigma'^*$

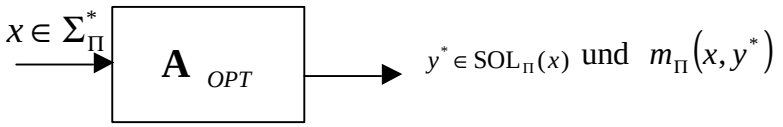


Mit $\mathbf{A}_f(x)$ wird das Berechnungsergebnis von $\mathbf{A}_f$ bei Eingabe von $x \in \Sigma_\Pi^*$ bezeichnet, d.h. $\mathbf{A}_f(x) = f(x)$.

Ein **Algorithmus $\mathbf{A}_{OPT}$ für ein Optimierungsproblem** Π mit Zielfunktion m_Π und Optimierungsziel $goal_\Pi \in \{\min, \max\}$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: $y^* \in \mathrm{SOL}_\Pi(x)$ und $m_\Pi(x, y^*) = goal_\Pi\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$



Mit $\mathbf{A}_{OPT}(x)$ wird die von $\mathbf{A}_{OPT}$ bei Eingabe von $x \in \Sigma_\Pi^*$ ermittelte Lösung bezeichnet, d.h. $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$.

1.3 Komplexitätsmaße

Die Komplexität eines Algorithmus wird in Abhängigkeit von der typischen **Größe der Eingabe** definiert. Je nach Problemstellung kann dieses beispielsweise sein:

- die Anzahl der Knoten eines Graphen

- die Anzahl zu sortierender Zahlen für einen Sortieralgorithmus
- die Dimension einer Matrix bei einem Invertierungsalgorithmus,
- die numerische Größe einer Zahl bei einem Algorithmus zur Primfaktorzerlegung,
- die Anzahl der Binärstellen zur Darstellung einer Zahl
- der Grad eines Polynoms.

Untersucht man das Laufzeitverhalten eines Algorithmus in Abhängigkeit von der Größe der Eingabe, übt die Definition des Begriffs „Problemgröße“ einen wesentlichen Einfluß aus, zumindest dann, wenn die Eingabe numerische Werte enthält. Beispielsweise führt die Prozedur

```
PROCEDURE zweier_potenz (    n : INTEGER;
                         VAR c : INTEGER);

VAR idx : INTEGER;

BEGIN {zweier-potenz }
  idx := n;               { Anweisung 1 }
  c   := 2;               { Anweisung 2 }
  WHILE idx > 0 DO        { Anweisung 3 }
    BEGIN
      c   := c*c;         { Anweisung 4 }
      idx := idx - 1;     { Anweisung 5 }
    END;
  c := c - 1;             { Anweisung 6 }
END   { zweier-potenz };
```

bei Eingabe einer Zahl $n > 0$ zur Berechnung des Ergebnisses $c = 2^{2^n} - 1$ insgesamt $3n + 4$ viele Anweisungen aus (hierbei werden nur die numerierten Anweisungen gezählt). Um das Ergebnis *c* beispielsweise im Binärsystem aufzuschreiben, werden 2^n viele binäre Einsen benötigt. Wählt man als Problemgröße den *Wert* der Eingabe, so würde diese Prozedur mit linearem Zeitaufwand (nämlich mit der Ausführung von $3n + 4$ vielen Anweisungen) ein Ergebnis von exponentieller Größe erzeugen. Diese Tatsache erscheint als nicht vernünftig; denn zur Erzeugung eines Objekts der Größe 2^n sollten auch mindestens 2^n viele Anweisungen durchgeführt werden müssen.

Wählt man als Problemgröße jedoch die *Anzahl der Binärstellen*, um die Eingabe *n* darzustellen, so hat die Eingabe die Größe $k = \lceil \log_2(n+1) \rceil$. Die Prozedur durchläuft $3n + 4 \geq 3 \cdot 2^{k-1} + 4 > 2^k$ viele Anweisungen. Die Eingabegröße auf diese Weise festzulegen, erscheint daher als vernünftig und nicht im Widerspruch zur Intuition.

Definiert man die Laufzeit eines Algorithmus bei einer gegebenen Eingabe als die Anzahl der von ihm durchlaufenen Anweisungen (**uniformes Kostenkriterium**), so legt man für den Fall, daß die Eingabe aus numerischen Daten besteht und die Laufzeit des Algorithmus vom

numerischen Wert der Eingabe abhängt, als Größe der Eingabe die Anzahl der Binärstellen fest, die benötigt werden, um die Eingabe darzustellen. Die Laufzeit des Algorithmus hängt zumindest dann von den numerischen Werten der Eingabe ab, wenn die Eingabewerte in arithmetischen Operationen vorkommen.

Für den Algorithmus **A** sei Σ^* die Menge der zulässigen Eingaben. Bei Eingabe von $x \in \Sigma^*$ bezeichnet $\mathbf{A}(x)$ die von **A** berechnete Lösung . Jedem $x \in \Sigma^*$ werde durch $size(x) \in \mathbf{N}$ eine (**Problem)- Größe** zugeordnet. Dann sei

$t_{\mathbf{A}}(x) =$ Anzahl der Anweisungen, die von **A** zur Berechnung von $\mathbf{A}(x)$ durchlaufen werden.

Die **Zeitkomplexität** von **A** (**im schlechtesten Fall**, **worst case**) wird definiert durch
$T_{\mathbf{A}}(n) = \max\{t_{\mathbf{A}}(x) \mid x \in \Sigma^* \text{ und } size(x) \le n\}$.

$T_{\mathbf{A}}(n)$ liefert eine Garantie für das Laufzeitverhalten von **A** bei Eingaben bis zur Größe *n*.

Die **Speicherplatzkomplexität** von **A** (**im schlechtesten Fall**, **worst case**) wird entsprechend definiert, wobei der Begriff „Anzahl der Anweisungen“ (Zeitverbrauch) durch den Begriff „Anzahl der Speicherplätze, die während der Berechnung maximal gleichzeitig belegt werden“ ersetzt wird.

Die **Zeitkomplexität** von **A im Mittel (average case**) wird folgendermaßen definiert:

Es wird ein Wahrscheinlichkeitsmaß *P* auf den zulässigen Eingaben von **A** festgelegt. Die mittlere Zeitkomplexität von **A** ist

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}[t_{\mathbf{A}}(x) \mid size(x) = n] = \int_{size(x)=n} t_{\mathbf{A}}(x) \cdot dP(x)$$

Meist ist man nicht am exakten Wert der Laufzeit eines Algorithmus interessiert, sondern nur an der **Größenordnung der Laufzeit** in Abhängigkeit von der Größe der Eingabe. Die Laufzeit des Algorithmus auf einem Rechner hängt ja eventuell von technischen Charakteristika des Rechners, eventuell von der verwendeten Programmiersprache, von der Qualität des Compilers oder von anderen Randfaktoren ab. Wesentlich ist jedoch, ob der Algorithmus beispielsweise ein exponentielles oder lineares Laufzeitverhalten aufweist. Ein Algorithmus $\mathbf{A_1}$ mit Zeitkomplexität $T_{\mathbf{A}_1}(n) = \frac{n \cdot (n+1)}{2}$ hat im wesentlichen dieselbe Zeitkomplexität wie ein Algorithmus $\mathbf{A_2}$ mit Zeitkomplexität $T_{\mathbf{A}_2}(n) = 3n^2 + 100n + 10$, nämlich quadratische Laufzeit, aber nicht wie ein Algorithmus $\mathbf{A_3}$ mit $T_{\mathbf{A}_3}(n) = 2^n + 1$. Von Interesse ist daher die **Größenordnung der Zeitkomplexität eines Algorithmus**.

Bei der Analyse des Laufzeitverhaltens eines Algorithmus im ungünstigsten Fall (worst case) wird man meist eine obere Schranke $g(n)$ für $T_{\mathbf{A}}(n)$ herleiten, d.h. man wird eine Aussage der Form $T_{\mathbf{A}}(n) \le g(n)$ begründen. In diesem Fall ist $T_{\mathbf{A}}(n) \in O(g(n))$. Eine zusätzliche Aussage $T_{\mathbf{A}}(n) \le h(n) \le g(n)$ mit einer Funktion $h(n)$ führt auf die verbesserte Abschätzung $T_{\mathbf{A}}(n) \in O(h(n))$. Man ist also bei der worst case-Analyse an einer möglichst kleinen oberen Schranke für $T_{\mathbf{A}}(n)$ interessiert.

Ein **Problem** Π **hat die worst case- (obere) Zeitkomplexität** $O(g(n))$, wenn es einen Lösungsalgorithmus $\mathbf{A}_{\Pi}$ für Π gibt mit $T_{\mathbf{A}_{\Pi}}(n) \in O(g(n))$.

Ein **Problem** Π **hat die Mindest- (untere) Zeitkomplexität** $\Omega(g(n))$, wenn für jeden Lösungsalgorithmus $\mathbf{A}_{\Pi}$ für Π $T_{\mathbf{A}_{\Pi}}(n) \in \Omega(g(n))$ gilt.

Ein **Problem** Π **hat die exakte Zeitkomplexität** $\Theta(g(n))$, wenn die worst case-Zeitkomplexität $O(g(n))$ und die Mindest-Zeitkomplexität $\Omega(g(n))$ ist.

Entsprechendes gilt für die **Speicherplatzkomplexität**.

2 Designtechniken

Im folgenden werden einige **wichtige Designtechniken zur Lösung eines Problems** im allgemeinen beschrieben und in den Unterkapiteln exemplarisch behandelt. Hierbei ist mit dem Begriff „Problem der Größe *n*“ eine Instanz der Größe *n* eines Problems gemeint.

1. Divide-and-Conquer (Teile-und-Herrsche)

Gegeben sei ein Problem der Größe *n*. Man führt man die folgenden Schritte durch:

1. ***Divide:*** Man zerlegt das Problem in eine Anzahl kleinerer disjunkter Teilprobleme (vom gleichen Problemtyp).

2. ***Conquer:*** Sind die Teilprobleme klein genug, so löst man diese direkt. Andernfalls löst man jedes Teilproblem (rekursiv) nach dem Divide-and-Conquer-Prinzip, d.h. mit demselben Algorithmus.

3. ***Combine:*** Man setzt die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammen.

Die Anwendung des Divide-and-Conquer-Prinzips setzt voraus, daß das Gesamtproblem auf geeignete Weise in mindestens zwei kleinere Teilprobleme zerlegt werden kann, die leichter zu lösen sind. Dies ist häufig dann möglich, wenn der Aufwand zur Lösung eines Problems lediglich von der Problemgröße und nicht von den Werten der Eingabegrößen einer Problemstellung abhängt. Eine weitere Voraussetzung ist die effiziente Kombinierbarkeit der Teillösungen zu einer Gesamtlösung des Problems.

2. Greedy-Methode (Optimierungsaufgaben):

Eine (global) optimale Lösung wird schrittweise gefunden, indem Entscheidungen gefällt werden, die auf „lokalen“ Informationen beruhen. Für eine Entscheidung wird hierbei nur ein kleiner (lokaler) Teil der Informationen genutzt, die über das gesamte Problem zur Verfügung stehen.

Es wird die in der jeweiligen Situation optimal erscheinende Entscheidung getroffen, unabhängig von vorhergehenden und nachfolgenden Entscheidungen. Die einmal getroffenen Entscheidungen werden im Laufe des Rechenprozesses nicht mehr revidiert.

Man geht nach folgendem Prinzip vor:

1. Man setzt $T := \emptyset$. Alle Elemente, die potentiell zu einer optimalen Lösung gehören können, gelten als „nicht behandelt“.

2. Solange es noch nicht behandelte Elemente gibt, wählt man ein Element *a* der noch nicht behandelten Elemente so aus, daß $T \cup \{a\}$ eine optimale Lösung bzgl. $T \cup \{a\}$ darstellt, und fügt *a* zu *T* hinzu. Das Element *a* gilt als „behandelt“.

3. *T* bestimmt die optimale Lösung.

Die Greedy-Strategie ist nicht für jedes Problem anwendbar, da u.U. aus dem Optimum einer Teillösung nicht auf das globale Optimum geschlossen werden kann.

3. Dynamische Programmierung (Optimierungsaufgaben):

Die Methode der dynamischen Programmierung stellt eine Verallgemeinerung des Divide-and-Conquer-Prinzips dar. Es ist auch dann anwendbar, wenn die Teilprobleme bei der Zerlegung nicht disjunkt sind, sondern wiederum gemeinsame Teilprobleme enthalten.

Wie beim Divide-and-Conquer-Prinzip wird ein Problem wird in mehrere kleinere (nun nicht notwendigerweise disjunkte) Teilprobleme aufgeteilt. Diese werden gelöst und aus deren Lösung eine Lösung für das Ausgangsproblem konstruiert. Dabei wird jedes Teilproblem nur einmal gelöst und das Ergebnis in einer Tabelle solange aufbewahrt, wie es eventuell später noch benötigt wird. Wiederholungen von Berechnungen werden so vermieden.

Die dynamische Programmierung ist dann anwendbar, wenn für das zu lösende Problem das folgende **Optimalitätsprinzip** gilt:

Jede Teillösung einer optimalen Lösung, die Lösung eines Teilproblems ist, ist selbst eine optimale Lösung des betreffenden Teilproblems.

Man geht nach folgenden Regeln vor:

1. Das zu lösende Problem wird rekursiv beschrieben.

2. Es wird die Menge **K** der kleineren Probleme bestimmt, auf die bei Lösung des Problems direkt oder indirekt zugegriffen wird.

3. Es wird eine Reihenfolge P_1, ..., P_r der Probleme in **K** festgelegt, so daß bei der Lösung von Problemen P_l $(1 \leq l \leq r)$ nur auf Probleme P_k mit Index k kleiner als l zugegriffen wird.

4. Die Lösungen für P_1, ..., P_r werden in dieser Reihenfolge berechnet und gespeichert, wobei einmal berechnete Lösungen solange gespeichert werden, wie sie für später noch zu berechnende Problemen direkt oder indirekt benötigt werden.

4. Aufzählungsmethoden: Backtrack-Methode:

Die Backtrack-Methode ist auf Probleme anwendbar, für deren Lösung kein besseres Verfahren bekannt ist, als alle möglichen Lösungskandidaten zu inspizieren und daraufhin zu untersuchen, ob sie als Lösung in Frage kommen. Die Backtrack-Methode organisiert diese erschöpfende Suche in einem im allgemeinen sehr großen Problemraum. Dabei wird ausgenutzt, daß sich oft nur partiell erzeugt Lösungskandidaten schon als inkonsistent ausschließen lassen.

Die effiziente Einsatz der Backtrack-Methode setzt voraus, daß das zu lösende Problem folgende Struktur aufweist:

1. Die Lösung ist als Vektor ($a[1], a[2], ...$) unbestimmter, aber endlicher Länge darstellbar.

2. Jedes Element $a[i]$ ist eine Möglichkeit aus einer endlichen Menge $A[i]$.

3. Es gibt einen effizienten Test zur Erkennung von inkonsistenten Teillösungen, d.h. Kandidaten ($a[1], a[2], ..., a[i]$), die sich zu keiner Lösung ($a[1], a[2], ..., a[i], a[i+1], ...$) erweitern lassen.

Das Verfahren kann allgemein so formuliert werden:

Schritt 1: Man wählt als erste Teillösung $a[1]$ ein mögliches Element aus $A[1]$.

Schritt n: Ist eine Teillösung ($a[1], a[2], ..., a[i]$) noch keine Gesamtlösung, dann erweitert man sie mit dem nächsten nicht inkonsistenten Element $a[i+1]$ aus $A[i+1]$ zur neuen Teillösung ($a[1], a[2], ..., a[i], a[i+1]$). Falls alle nicht inkonsistenten Elemente aus $A[i+1]$ bereits abgearbeitet sind, ohne daß man eine so erweiterte Teillösung gefunden wurde, geht man zurück und wählt $a[i]$ aus $A[i]$ neu (bzw. $a[i-1]$ aus $A[i-1]$ usw., wenn auch alle nicht inkonsistenten Kandidaten für $a[i]$ bereits abgearbeitet sind).

4. **Aufzählungsmethoden: Branch-and-Bound-Methode (Optimierungsaufgaben):**

Potentielle, aber nicht unbedingt optimale Lösungen werden systematisch erzeugt, z.B. mit der Backtrack-Methode. Diese werden in Teilmengen aufgeteilt, die sich auf Knoten eines Entscheidungsbaums abbilden lassen. Es wird eine Abschätzung für das Optimum mitgeführt und während des Verfahrensverlaufs laufend aktualisiert. Potentielle Lösungen, deren Zielfunktion einen „weit von der Abschätzung entfernten Wert“ aufweisen, werden nicht weiter betrachtet, d.h. der Teilbaum, der bei einer derartigen Lösung beginnt, muß nicht weiter durchlaufen werden.

Im folgenden werden die Methoden an einer Reihe von Algorithmen für unterschiedliche Probleme erläutert. Wie in Kapitel 1.3 erläutert, ist eine sorgfältige Definition der Größe einer Eingabeinstanz zumindest dann erforderlich, wenn die Eingabe aus numerischen Daten besteht und die Laufzeit des Algorithmus vom numerischen Wert der Eingabe abhängt (insbesondere dann, wenn die Eingabewerte in arithmetischen Operationen vorkommen).

Meist besteht eine Eingabeinstanz I aus einer Menge von n Objekten (Zahlen, Kantengewichte auf den Knoten eines Graphen), die jeweils einen numerischen Wert haben. Ist m der maximale absolute numerische Wert in der Eingabeinstanz, so bietet sich als Größe der Eingabeinstanz daher $size(I) = k = n \cdot \lceil \log(m+1) \rceil$ an, d.h. ein Wert, der proportional zur Anzahl der Bits ist, die benötigt werden, um I darzustellen.

In der Praxis ist man hauptsächlich an Verfahren interessiert, die höchstens ein polynomielles Laufzeitverhalten in der Größe der Eingabe haben. Stellt sich für einen Algorithmus exponentielles Laufzeitverhalten heraus und dieses bereits in Abgängigkeit von der Anzahl n der Objekte in einer Instanz, so kann man als Größe der Eingabe auch $size(I) = n$ wählen. Ein in n polynomielles Laufzeitverhalten erfordert hingegen die Untersuchung des Einflusses der Größen der numerischen Werte.

2.1 Divide and Conquer

Das Divide-and-Conquer-Prinzip soll am Sortier- und am Suchproblem erläutert werden. In der Regel führt es auf elegante rekursiv formulierte Algorithmen.

Sortieren von Elementen, auf denen eine Ordnungsrelation besteht

Instanz: $x = \langle a_1, ..., a_n \rangle$

x ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und

$a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist. $size(x) = n$.

Lösung: Eine Umordnung $p : [1:n] \to [1:n]$ der Elemente in *x*, so daß $\langle a_{p(1)}, ..., a_{p(n)} \rangle$ nach aufsteigenden Werten der *key*-Komponente sortiert ist, d.h. es gilt: $a_{p(i)} \le a_{p(i+1)}$ für $i = 1, ..., n-1$.

Als Randbedingung gilt, daß die Sortierung innerhalb der Folge *x* erfolgen soll, also insbesondere nicht zusätzlicher Speicherplatz der Größenordnung $O(n)$ erforderlich wird (**internes Sortierverfahren**).

Zur algorithmischen Behandlung wird eine Instanz *x* der Größe *n* in einem Feld `a` gespeichert, das definiert wird durch

```
CONST n         = ...;       { Problemgröße }

TYPE  idx_bereich = 1..n;
      key_typ     = ...;     { Typ der key-Komponente            }
      info_typ    = ...;     { Typ der info-Komponente           }
      entry_typ   = RECORD
                      key  : key_typ;
                      info : info_typ
                    END;
      feld_typ    = ARRAY [idx_bereich] OF entry_typ;
```

Die folgende Prozedur `interchange` vertauscht zwei Einträge miteinander:

```
PROCEDURE interchange (VAR x, y : entry_typ);
{ die Werte von x und y werden miteinander vertauscht             }

VAR z : entry_typ;

BEGIN { interchange }
  z := x;
  x := y;
  y := z;
END   { interchange };
```

Der folgende Algorithmus `quicksort` beruht auf der Idee der Divide-and-Conquer-Methode und erzeugt die umsortierte Eingabe $\langle a_{p(1)}, ..., a_{p(n)} \rangle$:

Besteht *x* aus keinem oder nur aus einem einzigen Element, dann ist nichts zu tun. Besteht *x* aus mehr Elementen, dann wird aus *x* ein Element *e* (**Pivot-Element**) ausgewählt und das

Problem der Sortierung von *x* in zwei kleinere Probleme aufgeteilt, nämlich der Sortierung aller Elemente, die kleiner als *e* sind (das sei das Problem der Sortierung der Folge *x(*lower*)*), und die Sortierung der Elemente die größer oder gleich *e* sind (das sei das Problem der Sortierung der Folge *x(*upper*)*); dabei wird als „Raum zur Sortierung" die Instanz *x* verwendet. Zuvor wird *e* an diejenige Position innerhalb von *x* geschoben, an der es in der endgültigen Sortierung stehen wird. Das Problem der Sortierung der erzeugten kleineren Probleme *x(*lower*)* und *x(*upper*)* wird (rekursiv) nach dem gleichen Prinzip gelöst. Die Position, an der *e* innerhalb von *x* in der endgültigen Sortierung stehen wird und damit die kleineren Probleme *x(*lower*)* und *x(*upper*)* bestimmt, sind dadurch gekennzeichnet, daß gilt:
$k < e$ für alle $k \in x(\text{lower})$ und $k \geq e$ für alle $k \in x(\text{upper})$.

Sortieralgorithmus mit `quicksort`:

Eingabe: $x = \langle a_1, ..., a_n \rangle$ ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \leq a_j$ genau dann, wenn $key_i \leq key_j$ ist

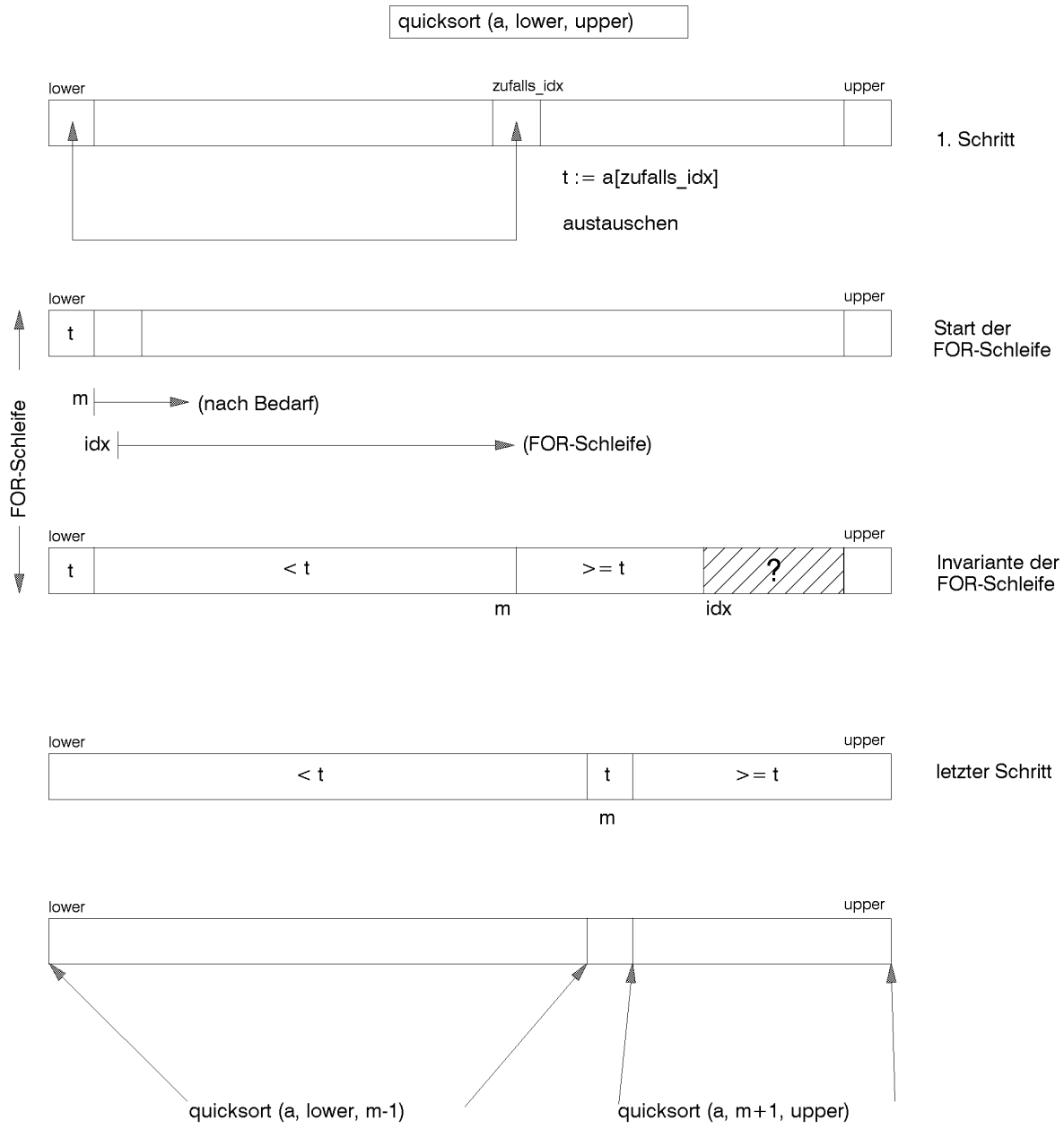
```
VAR a : feld_typ;          { Eingabefeld                }
    i : idx_bereich;

FOR i := 1 TO n DO
 BEGIN
   a[i].key   := key_i ;
   a.[i].info := info_i
 END;
```

Verfahren: Aufruf der Prozedur `quicksort (a, 1, n);`

Ausgabe: `a` mit `a[`i`].key` $\leq$ `a[`i `+ 1].key` für $i = 1, ..., n-1$

```
PROCEDURE quicksort (VAR a     : feld_typ;
                         lower : idx_bereich;
                         upper : idx_bereich);

     VAR  t           : entry_typ;
          m           : idx_bereich;
          idx         : idx_bereich;
          zufalls_idx : idx_bereich;

BEGIN { quicksort }
  IF lower < upper
  THEN BEGIN  { ein Feldelement zufällig aussuchen und
```

```
            mit a[lower] austauschen                          }
          zufalls_idx := lower
                         + Trunc(Random * (upper - lower + 1));
          interchange (a[lower], a[zufalls_idx]);

          t := a[lower];
          m := lower;

          FOR idx := lower + 1 TO upper DO
              { es gilt: a[lower+1] , ..., a[m] < t und
                         a[m+1], ... a[idx-1] >= t       }
              IF a[idx].key < t.key
              THEN BEGIN
                     m := m+1;
                     { vertauschen von a[m] und a[idx] }
                     interchange (a[m], a[idx])
                   END;

          { a[lower] und a[m] vertauschen }
          interchange (a[lower], a[m]);

          { es gilt: a[lower], ..., a[m-1] < a[m]  und
                         a[m] <= a[m+1], ..., a[upper]  }

          quicksort (a, lower, m-1);
          quicksort (a, m+1, upper);

        END { IF}

END { quicksort };
```


quicksort (a, lower, upper)
lower
zufalls_idx
upper
1. Schritt
t := a[zufalls_idx]
austauschen
lower
upper
t
Start der
FOR-Schleife
FOR-Schleife
m
(nach Bedarf)
idx
(FOR-Schleife)
lower
upper
t
< t
>= t
?
Invariante der
FOR-Schleife
m
idx
lower
upper
< t
t
>= t
letzter Schritt
m
lower
upper
quicksort (a, lower, m-1)
quicksort (a, m+1, upper)

Der Algorithmus stoppt, da in jedem `quicksort`-Aufruf ein Element an die endgültige Position geschoben wird und die Anzahl der Elemente in den `quicksort`-Aufrufen in den beiden restlichen Teilfogen zusammen um 1 verringert ist. Die Korrektheit des Algorithmus folgt aus der Invariante der `FOR`-Schleife.

Das beschriebene Verfahren mit der Prozedur `quicksort` sortiert eine Folge $x = \langle a_1, ..., a_n \rangle$ aus *n* Elementen mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n^2)$. Diese Schranke wird erreicht, wenn zufällig in jedem `quicksort`-Aufruf das Pivot-Element so gewählt wird, daß die Teilfolgen *x(*lower*)* kein Element und *x(*upper*)* alle restlichen Elemente enthalten (bzw. umgekehrt). Das Verfahren mit der Prozedur `quicksort` ist also von der Ordnung $\Theta(n^2)$. Im Mittel ist das Laufzeitverhalten jedoch wesentlich besser, nämlich von der Ordnung $O(n \cdot \log(n))$.

Es läßt sich zeigen, daß jedes Sortierverfahren, das auf dem Vergleich von Elementgrößen beruht, eine untere Zeitkomplexität der Ordnung $\Omega(n \cdot \log(n))$ und eine mittlere Zeitkomplexität ebenfalls der Ordnung $\Omega(n \cdot \log(n))$ hat, so daß das Verfahren mit der Prozedur `quicksort` optimales Laufzeitverhalten im Mittel aufweist. Daß das Verfahren mit der Prozedur `quicksort` in der Praxis ein Laufzeitverhalten zeigt, das fast alle anderen Sortierverfahren schlägt, liegt daran, daß die „ungünstigen“ Eingaben für `quicksort` mit gegen 0 gehender Wahrscheinlichkeit vorkommen und die Auswahl des Pivot-Elements zufällig erfolgt.

Ein Sortierverfahren, das auch im schlechtesten Fall (und im Mittel) die optimale Zeitkomplexität der Ordnung $O(n \cdot \log(n))$ hat, ist die Sortierung mit der Prozedur `heapsort`[2]. Diese beruht auf der Manipulation einer für das Sortierproblem geeigneten Datenstruktur, einem Binärbaum mit „Heapeigenschaft”, und nicht auf dem Divide-and-Conquer-Prinzip, so daß sie hier nur der Vollständigkeit halber angeführt wird.

[2] In der hier beschrieben Implementierung ist `heapsort` in der Praxis der Prozedur `quicksort` unterlegen. Varianten der Prozedur `heapsort` zeigen jedoch das theoretisch beweisbare gegenüber dem `quicksort` günstigere Laufzeitverhalten; siehe dazu: Wegener, I.: Bekannte Sortierverfahren und eine HEAPSORT-Variante, die QUICKSORT schlägt, Informatik-Spektrum, 13: 321-330, 1990.

```
PROCEDURE heapsort ( VAR a     : feld_typ;
                         lower : idx_bereich;
                         upper : idx_bereich);

    VAR idx : idx_bereich;

    PROCEDURE reconstruct_heap (VAR a : feld_typ;
                                i, j : idx_bereich);

    { Die Prozedur ordnet die Elemente im Feld a[i], ..., a[j]
      so um, daß anschließend die Heap-Eigenschaft gilt:
      a[k].key >= a[2k].key   für i <= k <= j/2 und
      a[k].key >= a[2k+1].key für i <= k <  j/2                  }

    VAR idx : idx_bereich;

    BEGIN { reconstruct_heap }
      IF i <= j/2    { d.h. a[i] ist kein Blatt }
      THEN BEGIN
             IF (a[i].key < a[2*i].key)
                 OR ( (2*i+1 <= j) AND (a[i].key < a[2*i+1].key) )
             THEN BEGIN { a[i] hat einen Sohn mit größerem Wert   }
                    IF 2*i+1 > j
                    THEN idx := 2*i
                    ELSE BEGIN
                           IF a[2*i].key >= a[2*i+1].key
                           THEN idx := 2*i
                           ELSE idx := 2*i+1;
                         END;
                    { Vertauschen von a[i] mit a[idx] }
                     interchange (a[idx], a[i]);
                    { die Heap-Eigenschaft im Teilbaum wieder
                      herstellen                                }
                     reconstruct_heap (a, idx, j)
                  END
             END
    END { reconstruct_heap } ;


BEGIN { heapsort }

      { buildheap:
        wandle a[lower] , ... a[upper] in einen Heap um;
        dabei wird ausgenutzt, daß die Elemente mit "hohen"
        Indizes keine Nachfolger haben                          }

      FOR idx := upper DIV 2 DOWNTO lower DO
         reconstruct_heap (a, idx, upper);
```

```
      { sortiere den Heap }
      idx := upper;
      WHILE idx >= 2 DO
      BEGIN
        { vertausche a[lower] mit a[idx] }
        interchange (a[lower], a[idx]);
        idx := idx - 1;
        reconstruct_heap (a, lower, idx)
      END   { WHILE }

END   { heapsort };
```

Suchen eines Elements in einer sortierten Folge

Instanz: $x = (\langle a_1, ..., a_n \rangle, a)$

$\langle a_1, ..., a_n \rangle$ ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist. Außerdem ist die Folge $\langle a_1, ..., a_n \rangle$ aufsteigend sortiert, d.h. es gilt: $a_i \le a_{i+1}$ für $i = 1, ..., n-1$; a ist ein Element der Form $a = (key, info)$.

$size(x) = n$.

Lösung: Entscheidung „ja“, wenn a in der Folge $\langle a_1, ..., a_n \rangle$ vorkommt; in diesem Fall soll auch die Position (der Index) von a innerhalb von $\langle a_1, ..., a_n \rangle$ ermittelt werden.

Entscheidung „nein“, wenn a in der Folge $\langle a_1, ..., a_n \rangle$ nicht vorkommt.

Bemerkung: $a = (key, info)$ kommt in der Folge $\langle a_1, ..., a_n \rangle$ genau dann vor, wenn die Folge ein Element $a_i = (key_i, info_i)$ enthält mit $key = key_i$.

Zur algorithmischen Behandlung wird eine Instanz x der Größe n (mit den obigen Deklarationen) in einem Feld `t` gespeichert. Der folgende Algorithmus `bin_search` beruht wieder auf der Idee der Divide-and-Conquer-Methode: Wenn die Eingabefolge $\langle a_1, ..., a_n \rangle$ leer ist, dann ist a in ihr nicht enthalten, und die Entscheidung lautet „nein“. Andernfalls wird das mittlere Element in $\langle a_1, ..., a_n \rangle$ daraufhin untersucht, ob die jeweiligen *key*-Komponenten übereinstimmen (bei einer geraden Anzahl von Elementen wird das erste Element der zweiten Hälfte genommen). Falls dieses zutrifft, ist die Suche beendet und die Position von a in $\langle a_1, ..., a_n \rangle$ bestimmt. Falls dieses nicht zutrifft, liegt a, wenn es überhaupt in $\langle a_1, ..., a_n \rangle$ vorkommt, im vorderen Abschnitt (nämlich dann, wenn die *key*-Komponente von a kleiner als die *key*-Komponente des Vergleichselements ist) bzw. im hinteren Abschnitt (nämlich dann, wenn die

key-Komponente von *a* größer als die *key*-Komponente des Vergleichselements ist). Die Entscheidung, in welchem Abschnitt weiterzusuchen ist, kann jetzt getroffen werden. Gleichzeitig wird durch diese Entscheidung die Hälfte aller potentiell noch auf Übereinstimmung mit *a* zu überprüfenden Elemente in $\langle a_1, ..., a_n \rangle$ ausgeschlossen. Im Abschnitt, der weiter zu überprüfen ist, wird nach dem gleichen Prinzip (rekursiv) verfahren. Unter Umständen muß die Suche fortgesetzt werden, bis ein noch zu überprüfender Abschnitt innerhalb $\langle a_1, ..., a_n \rangle$ nur noch ein Element enthält.

Suchalgorithmus mit Binärsuche:

Eingabe: $x = (\langle a_1, ..., a_n \rangle, a)$;

$\langle a_1, ..., a_n \rangle$ ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist. Außerdem ist die Folge $\langle a_1, ..., a_n \rangle$ aufsteigend sortiert, d.h. es gilt: $a_i \le a_{i+1}$ für $i = 1, ..., n-1$; *a* ist ein Element der Form $a = (key, info)$.

```
VAR t        : feld_typ;        { Eingabefeld          }
    a        : entry_typ;       { Suchelement          }
    gefunden : BOOLEAN;         { Entscheidung         }
    index    : idx_bereich;     { Position des Such-
                                  elements innerhalb der
                                  Eingabefolge         }
    i        : idx_bereich;

FOR i := 1 TO n DO
 BEGIN
   t[i].key  := key_i ;
   t[i].info := info_i
 END;

a[i].key  := key ;
a[i].info := info ;
```

Verfahren: Aufruf der Prozedur `bin_search (a, t, 1, n, gefunden, index);`

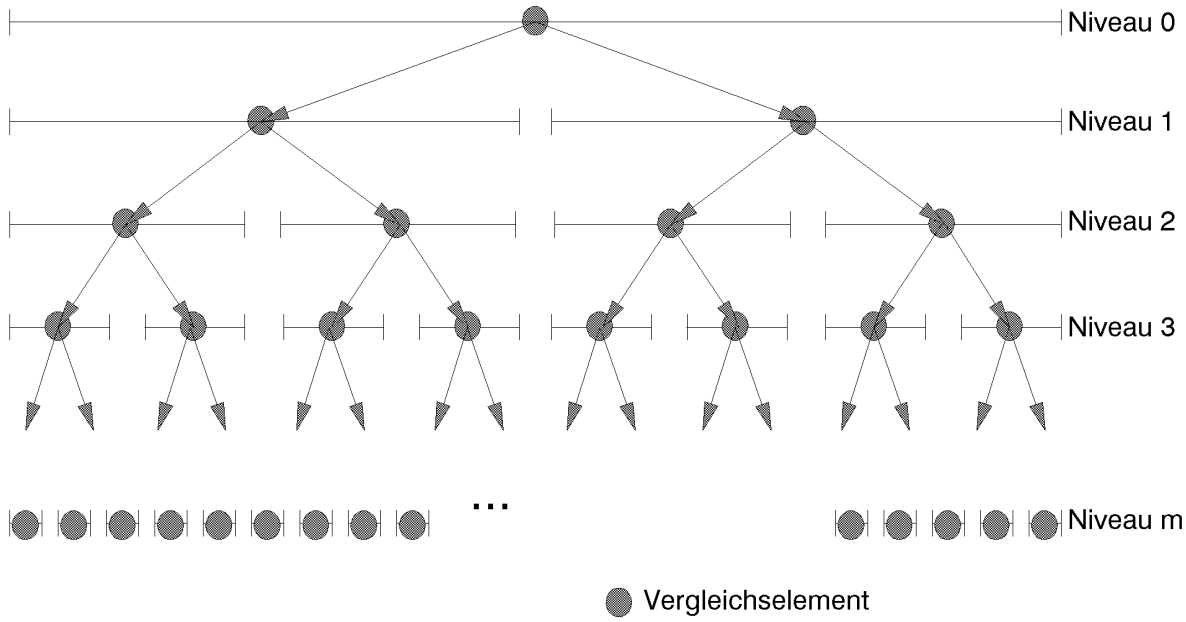
Ausgabe: `gefunden = TRUE`, `index = i`, falls *a* an der Position *i* in $\langle a_1, ..., a_n \rangle$ vorkommt; ansonsten `gefunden = FALSE`.

```
PROCEDURE bin_search (a           : entry_typ;
                      t           : feld_typ;
                      i_min       : idx_bereich;
                      i_max       : idx_bereich;
                      VAR gefunden: BOOLEAN;
                      VAR index   : idx_bereich);

{ gefunden = TRUE, falls a unter t[i_min], ..., t[i_max] vorkommt;
  in diesem Fall ist a = t[index] mit i_min ≤ index ≤ i_max;
  gefunden   = FALSE sonst                                        }

     VAR middle: idx_bereich; {Index auf das mittlere Feldelement
                               von t[i_min], ... t[i_max]         }


BEGIN { bin_search }
  gefunden := FALSE;
  IF i_min < i_max
  THEN BEGIN { der Feldausschnitt t[i_min], ..., t[i_max]
               enthält mindestens 2 Elemente                      }
         middle := i_min + ((i_max - i_min + 1) DIV 2);
         IF a.key = t[middle].key
         THEN BEGIN
                gefunden := TRUE;
                index    := middle
              END
         ELSE BEGIN
                IF a.key < t[middle].key
                THEN bin_search (a, t, i_min, middle-1,
                                                  gefunden, index)
                ELSE bin_search (a, t, middle + 1, i_max,
                                                  gefunden, index)
              END
       END
  ELSE BEGIN
         IF a.key = t[i_min].key
         THEN BEGIN
                gefunden := TRUE;
                index    := i_min
              END
       END
END   { bin_search };
```

Der Algorithmus stoppt, da in jedem `bin_search`-Aufruf, der einen Abschnitt der Länge k untersucht (k = `i_max - i_min + 1`), höchstens ein weiterer `bin_search`-Aufruf mit $\lfloor k/2 \rfloor$ vielen Elementen erfolgt. Die Korrektheit des Algorithmus ist offensichtlich.

Das beschriebene Verfahren mit der Prozedur `bin_search` entscheidet die Frage, ob ein Element *a* in einer sortierten Folge $x = \langle a_1, ..., a_n \rangle$ aus $n \geq 1$ Elementen vorkommt mit höchstens $\lfloor \log_2(n) \rfloor + 1$ vielen Elementvergleichen. Diese (worst-case-) Zeitkomplexität der Ordnung $O(\log(n))$ ist optimal.

Die folgende Abbildung zeigt alle möglichen Vergleichselemente, die im ungünstigten Fall inspiziert werden müssen.


Niveau 0
Niveau 1
Niveau 2
Niveau 3
...
Niveau m
Vergleichselement

2.2 Greedy-Methode

Die Greedy-Methode wird in diesem Kapitel sowohl an einem typischen Maximierungsproblem als auch an einem Minimierungsproblem erläutert.

Das Rucksackproblem als Maximierungsproblem (maximum knapsack problem)

Instanz: 1. $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i

$size(I) = n$

2. $\text{SOL}(I) = \left\{ (x_1, ..., x_n) \mid 0 \le x_i \le 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$

3. für $(x_1, ..., x_n) \in \text{SOL}(I)$ ist die Zielfunktion definiert durch

$$m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$$

4. *goal* = max

Lösung: $(x_1^*, ..., x_n^*) \in \text{SOL}(I)$ mit

(1) $0 \le x_i^* \le 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

Für eine Instanz $I = (A, M)$ mit $\sum_{i=1}^{n} w_i \le M$ setzt man $x_i^* := 1$ für i = 1, ..., n. Daher kann man im folgenden $\sum_{i=1}^{n} w_i > M$ annehmen.

Zur Ermittlung einer optimalen (maximalen) Lösung werden die Objekte $a_i = (w_i, p_i)$ und die Rucksackkapazität M einer Instanz $I = (A, M)$ in geeignete Datenobjekte eines Algorithmus umgesetzt.

Der Algorithmus verwendet zur Speicherung der Eingabemenge A die Datentypen

```
CONST n = ...;       { Problemgröße }

TYPE  idx_bereich = 1..n;

      input_item_typ = RECORD
                         w : REAL    { Gewicht eines Objekts     };
                         p : REAL    { Wert eines Objekts        };
                         j : INTEGER { ursprüngliche Objektnummer}
                       END;

      input_feld_typ = ARRAY[idx_bereich] OF input_item_typ;
```

Im Verlauf des Algorithmus werden die Objekte nach absteigenden Werten p_i/w_i umsortiert. Um die ursprüngliche Numerierung der Objekte zu ihrer Identifizierung zu erhalten, wird diese in der Komponente `j` eines Eingabeobjekts vom Typ `input_item_typ` vermerkt.

Algorithmus zur Lösung des Rucksackproblems:

Eingabe: $A = \{a_1, ..., a_n\}$ eine Menge von *n* Objekten und $M \in \mathbf{R}_{>0}$, die Rucksackkapazität. Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i

```
VAR A : input_feld_typ;   { Objekte                }
    M : REAL;             { Rucksackkapazität      }
    x : input_feld_typ;   { optimale Lösung        }
    i : idx_bereich;

FOR i := 1 TO n DO
  BEGIN
    A[i].w := w_i;
    A[i].p := p_i;
    A[i].j := i;
  END;
M := M;
```

Verfahren: Aufruf der Prozedur `greedy_rucksack (A, M, x);`

Ausgabe: x_i = `x[i]` für *i* = 1, ..., *n*.

```
PROCEDURE greedy_rucksack ( A     : input_feld_typ;
                            M     : REAL;
                            VAR x : input_feld_typ);

  { die Prozedur löst das Rucksackproblem für die Objekte im
    Feld A und die Rucksackkapazität M                          }

  VAR idx     : idx_bereich;    { Laufindex                       }
      current : REAL;           { verbleibende Rucksackkapazität  }
      y       : input_feld_typ; { Lösung der umsortierten Eingabe }

  BEGIN { greedy_rucksack }

    { Lösung mit 0 initialisieren }
    FOR idx := 1 TO n DO y[idx] := 0;

    { Umsortierung des Felds A nach absteigenden Werten p_i/w_i;
       das Ergebnis des umsortierten Felds steht wieder im
      Feld A; die Komponente A[idx].j wird nicht verändert.
      Pseudocode:                                               }
    -- SORT (A, absteigend nach Werten A[idx].p/A[idx].w);

    current := M;

    idx := 1;
    WHILE (idx <= n) AND (current > 0) DO
      BEGIN
        IF A[idx].w <= current
        THEN BEGIN
               y[idx] := 1;
               current := current - A[idx].w;
             END
        ELSE BEGIN { Rucksack gefüllt, keine weiteren Elemente
                     hinzunehmen                                }
               y[idx]  := current/A[idx].w;
               current := 0;
             END;
        idx := idx + 1;
      END;

     { Das Ergebnis in der ursprünglichen Numerierung zurückgeben:}
     FOR idx := 1 TO n DO
       x[A[idx].j] := y[idx];

  END { greedy_rucksack };
```

Es gilt:

Ist $I = (A, M)$ eine Instanz des Rucksackproblems (als Maximierungsproblem) mit $size(I) = n$, dann liefert das beschriebene Verfahren mit der Prozedur `greedy_rucksack` eine optimale Lösung mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n \cdot \log(n) + n)$. Sind die Eingabeobjekte bereits nach absteigenden Werten p_i / w_i sortiert, so kann man eine optimale Lösung mit der (worst-case-) Zeitkomplexität $O(n)$ ermitteln.

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

$size(G) = n$.

Lösung: Die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Zur algorithmischen Behandlung wird der Graph $G = (V, E, w)$ in Form seiner **Adjazenzmatrix** $A(G)$ gespeichert. Diese ist definiert durch

$$A(G) = A_{(n,n)} = \lfloor a_{i,j} \rfloor_{(n,n)} \text{ mit } a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}.$$

Es werden folgende Datentypen verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden      }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      bit_feld      = ARRAY [knotenbereich] OF BOOLEAN;
      feld          = ARRAY [knotenbereich] OF REAL;
```

Die folgende Funktion ermittelt das Minimum zweier REAL-Zahlen:

```
FUNCTION min (a, b: REAL):REAL;

BEGIN { min }
  IF a <= b THEN min := a
            ELSE min := b;
END   { min };
```

Der folgende Algorithmus zur Lösung des Problems der Wege mit minimalem Gewicht in gerichteten Graphen realisiert das allgemeine Prinzip der Greedy-Methode: Zunächst gelten alle Knoten v_i bis auf den Knoten v_1 als „nicht behandelt“. Aus den nicht behandelten Knoten wird ein Knoten ausgewählt, der zu einer bisherigen Teillösung hinzugenommen wird, und zwar so, daß dadurch eine bzgl. der Teillösung, d.h. bzgl. der behandelten Knoten, optimale Lösung entsteht.

Eingabe: $G = (V, E, w)$ ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

```
VAR A       : matrix;        { Adjazenzmatrix }
    i       : knotenbereich;
    j       : knotenbereich;
    distanz : feld;

FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i, v_j) ∈ E THEN A[i, j] := w((v_i, v_j))
                      ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `minimale_wege (A, distanz);`

Ausgabe: $d_i =$ `distanz[i]` für $i = 1, ..., n$.

```
PROCEDURE minimale_wege ( A           : matrix;
                          VAR distanz : feld);

VAR rest_knoten : bit_feld;      { Kennzeichnung der noch nicht
                                   behandelten Knoten: ist v_i ein
                                   nicht behandelter Knoten, so
                                   ist rest_knoten[i] = TRUE      }
    idx         : knoten_bereich;
    u           : knoten_bereich;
    exist       : BOOLEAN;
```

```
PROCEDURE auswahl (VAR knoten   : knoten_bereich;
                   VAR gefunden : BOOLEAN);

  { die Prozedur wählt unter den noch nicht behandelten
    Knoten einen Knoten mit minimalem distanz-Wert aus
   (in knoten); gibt es einen derartigen Knoten, dann ist
    gefunden = TRUE, sonst ist gefunden = FALSE                }

    VAR  i   : knoten_bereich;
         min : REAL;

   BEGIN { auswahl }
     gefunden := FALSE;
     min      := unendlich;

     FOR i := 1 TO n DO
       IF rest_knoten [i]
       THEN BEGIN
              IF distanz [i] < min
              THEN BEGIN
                     gefunden := TRUE;
                     knoten   := i;
                     min      := distanz [i]
                   END
          END

    END { auswahl };


BEGIN { minimale_wege }
  { Gewichte zum Startknoten initialisieren und alle Knoten,
    bis auf den Startknoten, als nicht behandelt kennzeichnen: }
   FOR idx := 1 TO n DO
     BEGIN
       distanz[idx]     := A[1, idx];
       rest_knoten[idx] := TRUE
     END;
   distanz[1]     := 0;
   rest_knoten[1] := FALSE;

   { einen Knoten mit minimalem Distanzwert aus den Restknoten
     auswählen:                                                }
   auswahl (u, exist);

   WHILE exist DO
     BEGIN
       { den gefundenen Knoten aus den noch nicht behandelten
```

```
          Knoten herausnehmen:                                    }
        rest_knoten[u] := FALSE;

        { für jeden noch nicht behandelten Knoten idx, der mit
          dem Knoten u verbunden ist, den distanz-Wert auf den
          neuesten Stand bringen:                                 }
        FOR idx := 1 TO n DO
          IF rest_knoten[idx]
             AND
             (A[u, idx] <> unendlich)
          THEN distanz [idx] := min (distanz[idx],
                                     distanz[u] + A[u, idx]);

        { einen neuen Knoten mit minimalem Distanzwert aus den
          Restknoten auswählen:                                   }
        auswahl (u, exist);

      END  {WHILE exist }

END { minimale_wege };
```

Es gilt bei Eintritt in die `WHILE exist DO`-Schleife folgende Schleifeninvariante:

(1) Ist der Knoten v_i ein bereits behandelter Knoten, d.h. `rest_knoten[i]=FALSE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

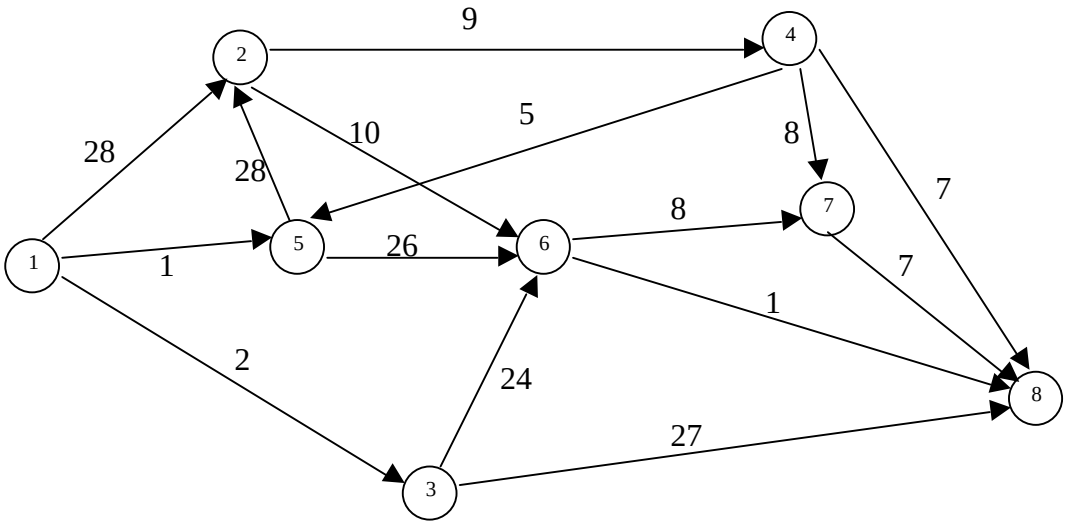
(2) Ist der Knoten v_i ein nicht behandelter Knoten, d.h. `rest_knoten[i]=TRUE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der bis auf v_i nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

Die Schleifeninvariante ermöglicht die Anwendung der Greedy-Methode; aus ihr folgt die Korrektheit des Algorithmus.

Ist $G=(V,E,w)$ eine Instanz des Problems der Wege mit minimalem Gewicht in gerichteten Graphen mit $size(G)=n$, dann liefert das beschriebene Verfahren mit der Prozedur `minimale_wege` im Feld `distanz` die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n^2)$.

Beispiel:

Die Instanz $G = (V, E, w)$ sei gegeben durch folgende graphische Darstellung:



Die Adjazenzmatrix zu *G* lautet

$$A(G) = \begin{bmatrix} \infty & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 9 & \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & \infty & 5 & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & \infty & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 8 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

`u` wird jeweils beim Aufruf der Prozedur `auswahl (u, exist)` ausgewählt:

`u`	nichtbehandelte Knoten	`distanz [1]`	`distanz [2]`	`distanz [3]`	`distanz [4]`	`distanz [5]`	`distanz [6]`	`distanz [7]`	`distanz [8]`
	2 3 4 5 6 7 8	---	28	2	∞	1	∞	∞	∞
5	2 3 4 6 7 8		9	2	∞	---	27	∞	∞
3	2 4 6 7 8		9	---	∞		26	∞	29
2	4 6 7 8		---		18		19	∞	∞
4	6 7 8				---		19	26	25
6	7 8						---	26	20
8	7							26	---
7	leer							---	

2.3 Dynamische Programmierung

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem (minimum traveling salesperson problem)

Instanz:

1. $G = (V, E, w)$

   $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht
2. $\text{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist eine Tour durch $G\}$;

   eine Tour durch *G* ist eine geschlossene Kantenfolge, in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt
3. für $T \in \text{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion definiert durch $m(G,T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$
4. $goal = \min$

Lösung: Eine Tour $T^* \in \text{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht, und $m(G, T^*)$.

Im folgenden wird $v_{i_1} = v_1$ gesetzt.

Um die Methode der Dynamischen Programmierung anzuwenden, wird das Problem rekursiv beschrieben (Regel 1.):

Das Problem wird allgemein mit $\Pi(v_i, S)$ bezeichnet. Dabei ist $v_i \in V$ und $S \subseteq V$ mit $v_i \notin S$ und $v_1 \notin S$. $\Pi(v_i, S)$ beschreibt das Problem der Bestimmung eines Weges mit minimalem Gewicht von v_i durch alle Knoten von *S* nach v_1. Das für $\Pi(v_i, S)$ ermittelte (optimale) Gewicht sei $g(v_i, S)$. Eventuell ist $g(v_i, S) = \infty$.

Gesucht wird eine Lösung von $\Pi(v_1, V - \{v_1\})$ bzw. $g(v_1, V - \{v_1\})$.

Eine rekursive Problembeschreibung ergibt sich aus der Überlegung, daß eine kostenoptimale Tour für das Problem $\Pi(v_i, S)$ mit einer Kante $(v_i, v_j) \in E$ mit $v_j \in S$ beginnt und dann aus einem kostenoptimalen Weg von v_j durch alle Knoten von $S - \{v_j\}$ zum Knoten v_1 besteht. Daher gelten die Rekursionsgleichungen für $g(v_i, S)$:

$g(v_i, S) = \min\{\, w((v_i, v_j)) + g(v_j, S - \{v_j\}) \mid v_j \in S \,\}$ für $v_i \notin S$ und $v_1 \notin S$

$g(v_i, \varnothing) = w((v_i, v_1))$ für $i = 2, ..., n$.

Diese Rekursionsformeln legen die Berechnungsreihenfolge fest; die Mengen auf der rechten Seite bei der Berechnung von g werden immer weniger mächtig (Regel 2. und Regel 3.):

Teilproblem (Berechnung in aufsteigender Reihenfolge)	Optimaler Wert der Zielfunktion
$\Pi(v_i, \varnothing)$ für $i = 2, ..., n$	$g(v_i, \varnothing)$ für $i = 2, ..., n$
$\Pi(v_i, S)$ mit $\lvert S\rvert = 1$, $v_i \notin S$, $v_1 \notin S$	$g(v_i, S)$ mit $\lvert S\rvert = 1$, $v_i \notin S$, $v_1 \notin S$
$\Pi(v_i, S)$ mit $\lvert S\rvert = 2$, $v_i \notin S$, $v_1 \notin S$	$g(v_i, S)$ mit $\lvert S\rvert = 2$, $v_i \notin S$, $v_1 \notin S$
...	...
$\Pi(v_i, S)$ mit $\lvert S\rvert = n-1$, $v_i \notin S$, $v_1 \notin S$, d.h. $\Pi(v_1, V - \{v_1\})$	$g(v_i, S)$ mit $\lvert S\rvert = n-1$, $v_i \notin S$, $v_1 \notin S$, d.h. $g(v_1, V - \{v_1\})$

Beispiel:

Gegeben sei der folgende Graph $G = (V, E, w)$ mit $n = 4$ Knoten durch seine Adjazenzmatrix

$$A(G) = \begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix}.$$

$g(v_2, \varnothing) = w((v_2, v_1)) = 5$, $g(v_3, \varnothing) = w((v_3, v_1)) = 6$, $g(v_4, \varnothing) = w((v_4, v_1)) = 8$

$g(v_1, \{v_2\}) = w((v_1, v_2)) + g(v_2, \varnothing) = 15$, $g(v_1, \{v_3\}) = w((v_1, v_3)) + g(v_3, \varnothing) = 21$,
$g(v_1, \{v_4\}) = w((v_1, v_4)) + g(v_4, \varnothing) = 28$,
$g(v_2, \{v_3\}) = w((v_2, v_3)) + g(v_3, \varnothing) = 15$, $g(v_2, \{v_4\}) = w((v_2, v_4)) + g(v_4, \varnothing) = 18$,
$g(v_3, \{v_2\}) = w((v_3, v_2)) + g(v_2, \varnothing) = 18$, $g(v_3, \{v_4\}) = w((v_3, v_4)) + g(v_4, \varnothing) = 20$,
$g(v_4, \{v_2\}) = w((v_4, v_2)) + g(v_2, \varnothing) = 13$, $g(v_4, \{v_3\}) = w((v_4, v_3)) + g(v_3, \varnothing) = 15$

$g(v_1, \{v_2, v_3\}) = \min\{w((v_1, v_2)) + g(v_2, \{v_3\}),\ w((v_1, v_3)) + g(v_3, \{v_2\})\} = 25$,
$g(v_1, \{v_2, v_4\}) = \min\{w((v_1, v_2)) + g(v_2, \{v_4\}),\ w((v_1, v_4)) + g(v_4, \{v_2\})\} = 28$,
$g(v_1, \{v_3, v_4\}) = \min\{w((v_1, v_3)) + g(v_3, \{v_4\}),\ w((v_1, v_4)) + g(v_4, \{v_3\})\} = 35$
$g(v_2, \{v_3, v_4\}) = \min\{w((v_2, v_3)) + g(v_3, \{v_4\}),\ w((v_2, v_4)) + g(v_4, \{v_3\})\} = 25$,

$$g(v_3,\{v_2,v_4\}) = \min\{w((v_3,v_2)) + g(v_2,\{v_4\}),\ w((v_3,v_4)) + g(v_4,\{v_2\})\} = 25,$$
$$g(v_4,\{v_2,v_3\}) = \min\{w((v_4,v_2)) + g(v_2,\{v_3\}),\ w((v_4,v_3)) + g(v_3,\{v_2\})\} = 23$$

$$g(v_1,\{v_2,v_3,v_4\}) = \min\{w((v_1,v_2)) + g(v_2,\{v_3,v_4\}),\ w((v_1,v_3)) + g(v_3,\{v_2,v_4\}),\ w((v_1,v_4)) + g(v_4,\{v_2,v_3\})\} = 3$$

Algorithmus zur Lösung des Problems des Handlungsreisenden:

Es werden wieder die bereits bekannten Deklarationen verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden       }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
```

Die Knotennummern einer optimalen Tour werden in der berechneten Reihenfolge in Form einer verketteten Liste ermittelt: der Typ eines Eintrags dieser Liste ergibt sich aus der Typdeklaration

```
TYPE tour_item_ptr = ^tour_item;
     tour_item     = RECORD
                        nr   : knoten_bereich;
                        next : tour_item_ptr  { Verweis auf die
                                    nächste  Knotennummer der Tour }
                     END;
```

Außerdem wird die Typdeklaration

```
TYPE knoten_menge = SET OF knoten_bereich;
```

verwendet.

Die folgende Funktion ermittelt das Minimum zweier REAL-Zahlen:

```
FUNCTION min (a, b: REAL):REAL;
```

Eingabe: $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

```
VAR A        : matrix;        { Adjazenzmatrix }
    i        : knotenbereich;
    j        : knotenbereich;
    tour_ptr : tour_item_ptr;
    laenge   : REAL;

FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i,v_j) ∈ E THEN A[i, j] := w((v_i,v_j))
                     ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `salesman (A, tour_ptr, laenge);` die Prozedur `salesman` ruft zur Ermittlung einer optimalen Tour die Prozedur `dyn_g` auf.

Ausgabe: `laenge` gibt das minimale Gewicht einer Tour durch G an, die beim Knoten v_1 beginnt und endet; `tour_ptr` verweist auf den ersten Eintrag einer verketteten Liste mit den Knotennummern einer optimalen Tour (in dieser Reihenfolge).

```
PROCEDURE salesman (A            : matrix;
                    VAR tour_ptr : tour_item_ptr;
                                   { Verweis auf den ersten
                                     Knoten der Tour        }
                    VAR laenge   : REAL  { Länge der Tour   }
                      );

  BEGIN { salesman }
    dyn_g(1, [2..n], n - 1, laenge, tour_ptr)
  END   {salesman };
```

Dabei ist dyn_g definiert durch:

```
PROCEDURE  dyn_g (i             : knoten_bereich;
                  s             : knoten_menge;
                  anz           : INTEGER; { Mächtigkeit von s  }
                  VAR wert      : REAL;    { minimales Gewicht  }
                  VAR start_ptr : tour_item_ptr;
                                  { Verweis auf den ersten Knoten
                                    einer Tour mit min. Gewicht,
                                    beginnend beim Knoten v_i, durch
                                    alle Knoten von s nach v_1     }
                           );

   VAR    k_idx    : knoten_bereich;
          k_next   : knoten_bereich;
          c_wert   : REAL;
          c_ptr    : tour_item_ptr;
          item_ptr : tour_item_ptr;

   BEGIN {dyn_g }

     New(item_ptr);
     item_ptr^.nr   := i;
     start_ptr      := item_ptr;

     IF anz = 0
     THEN BEGIN
            wert           := A[i, 1];
            item_ptr^.next := NIL
          END
     ELSE BEGIN
            wert := unendlich;
            FOR k_idx := 2 TO n DO
              BEGIN
               IF (k_idx IN s)
               THEN BEGIN
                       dyn_g (k_idx, s - [k_idx], anz - 1,
                                                  c_wert, c_ptr);
                       IF wert >=  A[i, k_idx] + c_wert
                       THEN BEGIN
                              wert := A[i, k_idx] + c_wert;
                              item_ptr^.next := c_ptr
                            END
                    END { IF }
              END  { FOR }
          END
   END   {dyn_g };
```

Die (worst-case-) Zeitkomplexität des Verfahrens zur Berechnung von $g(v_1, V - \{v_1\})$ bei Eingabe einer Instanz G mit $size(G) = n$ ist von der Ordnung $O(n^2 \cdot 2^n)$.

Bemerkung: Die exponentielle Laufzeit des Verfahrens ergibt sich bereits in Abhängigkeit von der Anzahl der Knoten der Eingabeinstanz. Bei einer Berücksichtigung der Werte der Kantengewichte in der Wahl für $size(G)$ bleibt es bei einem exponentiellen Laufzeitverhalten. Daher ist die Wahl $size(G) = n$ für eine Instanz G mit n Knoten sinnvoll.

Das 0/1-Rucksackproblem als Maximierungsproblem (maximum 0/1 knapsack problem)

Instanz: 1. $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i.

$size(I) = n \cdot \lceil \log(p_{max}) \rceil$ mit $p_{max} = \max\{p_i \mid i = 1, ..., n\}$

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$; man beachte, daß hier nur Werte $x_i = 0$ oder $x_i = 1$ zulässig sind

3. $m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \mathrm{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

Für eine Instanz $I = (A, M)$ wird im folgenden wieder $\sum_{i=1}^{n} w_i > M$ angenommen.

Die Greedy-Methode zur Lösung des (allgemeinen) Rucksackproblems führt, angewandt auf das 0/1-Rucksackproblem, nicht auf eine optimale Lösung, wie folgendes Beispiel zeigt:

$I = (A, M)$ mit $n = 3$,
$a_1 = (w_1, p_1) = (510, 550)$, $a_2 = (w_2, p_2) = (500, 500)$, $a_3 = (w_3, p_3) = (500, 500)$,
$M = 1000$
führt bei Anwendung der Greedy-Methode auf $x_1 = 1$, $x_2 = x_3 = 0$ und den Wert der Zielfunktion $m(I, (x_1, x_2, x_3)) = 550$. Zu beachten ist hierbei, daß die Objekte nicht geteilt werden dürfen. Eine optimale Lösung für *I* lautet jedoch $x_1^* = 0$, $x_2^* = x_3^* = 1$ mit $m^*(I) = 1000$.

Es sei die Instanz $I = (A, M)$ mit $size(I) = n$ gegeben. Die Menge *A* der Objekte sei in der folgenden Darstellung fest. Das Problem der Ermittlung einer optimalen Lösung für diese Instanz wird in kleinere Probleme zerlegt, die jeweils nur einen Teil der Objekte in *A* und variable Rucksackkapazitäten betrachten. Aus optimalen Lösungen dieser kleineren Probleme wird schrittweise eine optimale Lösung des ursprünglichen Problems zusammengesetzt.

Das ursprüngliche Problem der Ermittlung einer optimalen Lösung für *I* betrachtet alle *n* Elemente und die Rucksackkapazität *M*. Es soll daher mit *RUCK*(*n*, *M*) bezeichnet werden. Werden nur die Objekte $a_1, ..., a_j$ und die Rucksackkapazität *y* betrachtet, so wird dieses kleinere Problem mit *RUCK*(*j*, *y*) bezeichnet. Eine optimale Lösung für *RUCK*(*j*, *y*) habe einen Wert der Zielfunktion, der mit $f_j(y)$ bezeichnet wird.

Gesucht wird also neben $(x_1^*, ..., x_n^*) \in \mathrm{SOL}(I)$ der Wert $m^*(I) = f_n(M)$.

Bei der Methode der Dynamischen Programmierung wird das zu lösende Problem zunächst rekursiv beschrieben (Regel 1.), die Menge der kleineren Probleme bestimmt, auf die bei der Lösung des Problems direkt oder indirekt zugegriffen wird (Regel 2.), und die Reihenfolge festgelegt, in der diese Teilprobleme gelöst werden (Regel 3.):

Umsetzung der Regel 1.:

Zwischen den Problemen *RUCK*(*j*, *y*) mit ihren Zielfunktionswerten $f_j(y)$ besteht folgender (rekursiver) Zusammenhang:
$f_j(y) = \max\{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\}$ für $j = n, ..., 1, 0 \le y \le M$ („**Rückwärtsmethode**“)
$f_0(y) = 0$ für $0 \le y \le M$ und $f_0(y) = -\infty$ für $y < 0$.

$f_j(y)$ kann als Funktion von *y* aufgefaßt werden. Aus der Kenntnis des (gesamten) Funktionsverlaufs von f_n ist insbesondere $m^*(I) = f_n(M)$ ablesbar. Es zeigt sich (siehe unten), daß aus der Kenntnis des Funktionsverlaufs von f_n an der Stelle *M* auch eine optimale Lösung

$(x_1^*, ..., x_n^*) \in \text{SOL}(I)$ „leicht“ ermittelt werden kann. Daher wird zunächst eine Methode zur Berechnung und Speicherung der Funktionsverläufe $f_j(y)$ beschrieben.

Umsetzung der Regeln 2. und 3.:

Teilproblem (Berechnung in aufsteigender Reihenfolge)	Optimaler Wert der Zielfunktion
RUCK(0, *y*), $y \le M$	$f_0(y)$ mit $y \le M$
RUCK(1, *y*), $0 \le y \le M$	$f_1(y)$ mit $y \le M$
RUCK(2, *y*), $0 \le y \le M$	$f_2(y)$ mit $y \le M$
...	
RUCK(*n*, *y*), $0 \le y \le M$	$f_n(y)$ mit $y \le M$

Zur Berechnung von $f_j(y)$ gemäß der rekursiven Formel

$f_j(y) = \max\{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\}$ für $j = n, ..., 1, 0 \le y \le M$

braucht wegen $y \le M$ und $y - w_j \le M$ nur die Funktion $f_{j-1}(y)$ bereitgehalten zu werden. Um jedoch aus f_n an der Stelle *M* eine optimale Lösung $(x_1^*, ..., x_n^*) \in \text{SOL}(I)$ zu ermitteln, werden alle Funktionen $f_j(y)$ für *j* = 0, ..., *n* benötigt.

Man sieht leicht, daß wegen $x_i \in \{0,1\}$ und $p_i \in \mathbf{R}_{>0}$ die Funktionen $f_j(y)$ Treppenfunktionen sind. Zur Algorithmen-gerechten Speicherung des Funktionsverlaufs von $f_j(y)$ brauchen daher nur die Sprungstellen der Funktion gespeichert werden.

Es sei S_j für *j* = 0, ..., *n* die Menge der Sprungstellen von f_j.

Umsetzung der Regel 4.:

Die algorithmische Umsetzung des Lösungsverfahrens für das 0/1-Rucksackproblem mit Hilfe der Methode der Dynamischen Programmierung, d.h. die Erzeugung der Funktionen f_0, ..., f_n bzw. der Mengen ihrer Sprungstellen S_0, ..., S_n und die Ermittlung einer optimalen Lösung, wird im folgenden nur mit Hilfe von Pascal-ähnlichem Pseudocode angegeben. Dabei werden folgende Begriffe verwendet:

Ein Zahlenpaar (W, P) **dominiert** ein Zahlenpaar (W', P'), wenn $W \le W'$ und $P \ge P'$ gilt.

Für zwei Mengen *A* und *B* von Zahlenpaaren sei $A \otimes B$ die Vereinigungsmenge von A und B ohne die Paare aus *A*, die durch ein Paar aus B dominiert werden, und ohne die Paare aus *B*, die durch ein Paar aus *A* dominiert werden.

Algorithmus zur Lösung des 0/1-Rucksackproblems:

Eingabe: $A=\{a_1,...,a_n\}$ eine Menge von *n* Objekten und $M \in \mathbf{R}_{>0}$, die Rucksackkapazität. Jedes Objekt a_i, $i=1,...,n$, hat die Form $a_i=(w_i,p_i)$

Verfahren: { Berechnung der Funktionen f_j bzw. der Mengen S_j ihrer Sprungstellen für $j=1,...,n$ }

```
BEGIN
   S_0 := {(0,0)};

   FOR j := 1 TO n DO
      BEGIN
         Ŝ_j := {(W + w_j, P + p_j) | (W,P) ∈ S_{j-1}};
         S_j := S_{j-1} ⊗ Ŝ_j ;
      END;

END;
```

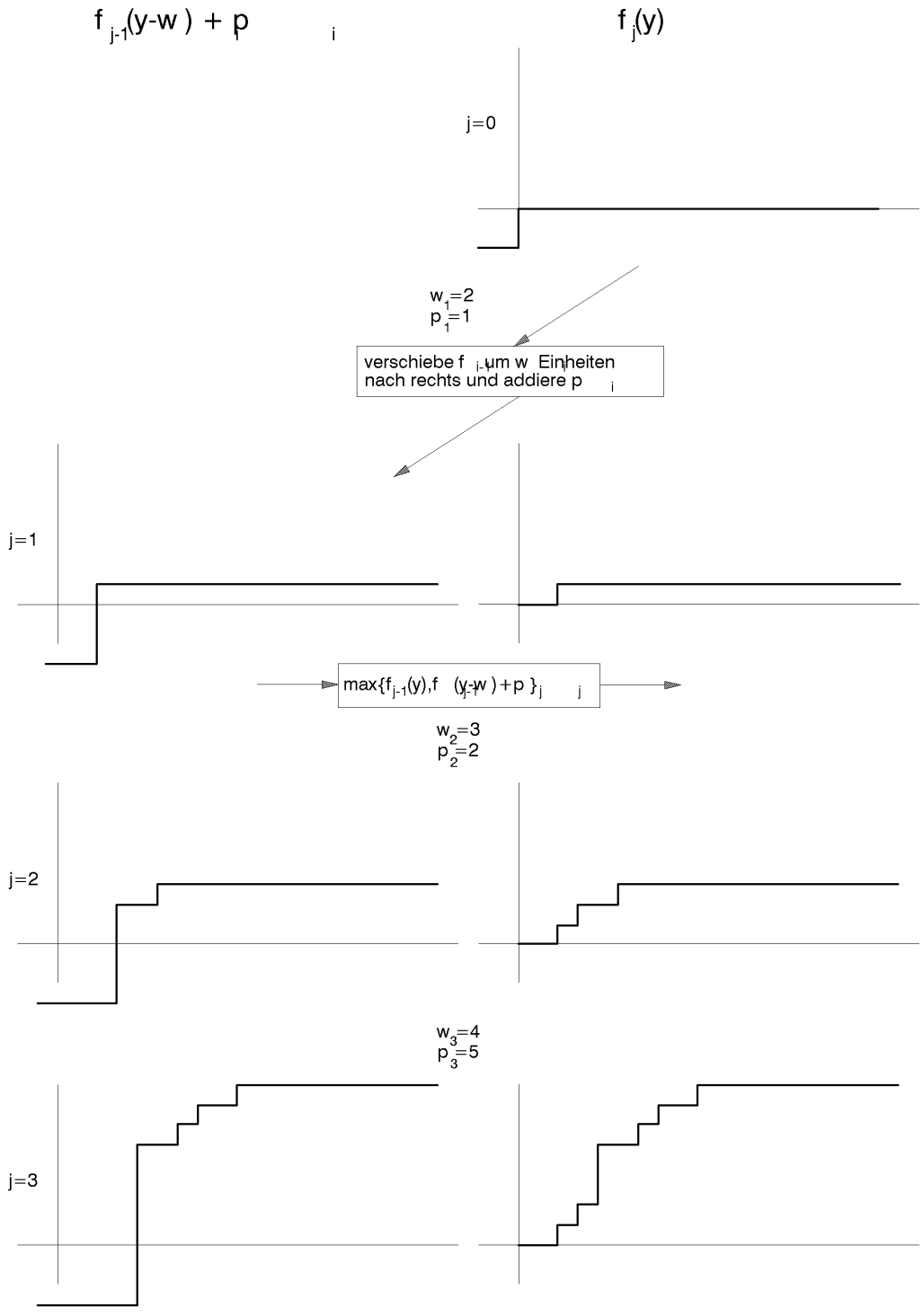
{ Die Paare in S_j seien aufsteigend nach der ersten (und damit auch nach der zweiten) Komponente geordnet, d.h. $S_j=\{(W_{0,j},P_{0,j}),...,(W_{i_j,j},P_{i_j,j})\}$ mit $W_{k,j} \le W_{k+1,j}$ für $0 \le k < i_j$.
Für $y \ge 0$ gelte $W_{k,j} \le y < W_{k+1,j}$. Dann ist $f_j(y)=P_{k,j}$. }

{ Berechnung einer optimierende Auswahl $x_1^*,...,x_n^*$ }

Es sei (W,P) dasjenige Paar in S_n mit $f_n(M)=P$.

```
FOR k := n DOWNTO 1 DO
   BEGIN
      IF (W,P) ∈ S_{k-1}
      THEN x_k^* := 0
      ELSE BEGIN
               x_k^* := 1;
               W := W - w_k ;
               P := P - p_k ;
           END;
   END;
```

Ausgabe: $(x_1^*, ..., x_n^*) \in \mathrm{SOL}(I)$ und der maximale Wert $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ der Zielfunktion.



Beispiel:

Eingabe $A = \{a_1, ..., a_7\}, ..., a_i = (w_i, p_i)$ für $i = 1, ..., 7$:

i	Gewicht w_i	Wert p_i
1	2	10
2	3	5
3	5	15
4	7	7
5	1	6
6	4	18
7	1	3

Rucksackkapazität $M = 15$

Berechnung der Funktionen f_j bzw. der Mengen S_j ihrer Sprungstellen für $j = 1, ..., n$:

$S_0 = \{(0,0)\}$

$\hat{S}_1 = \{(2,10)\}$ $\quad S_1 = \{(0,0),(2,10)\}$

$\hat{S}_2 = \{(3,5),(5,15)\}$ $\quad S_2 = \{(0,0),(2,10),(5,15)\}$

$\hat{S}_3 = \{(5,15),(7,25),(10,30)\}$

$S_3 = \{(0,0),(2,10),(5,15),(7,25),(10,30)\}$

$\hat{S}_4 = \{(7,7),(9,17),(12,22),(14,32),(17,37)\}$

$S_4 = \{(0,0),(2,10),(5,15),(7,25),(10,30),(14,32),(17,37)\}$

$\hat{S}_5 = \{(1,6),(3,16),(6,21),(8,31),(11,36),(15,38),(18,43)\}$

$S_5 = \{(0,0),(1,6),(2,10),(3,16),(6,21),(7,25),(8,31),(11,36),(15,38),(18,43)\}$

$\hat{S}_6 = \{(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(15,54),(19,56),(22,61)\}$

$S_6 = \{(0,0),(1,6),(2,10),(3,16),(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(15,54),(19,56),(22,61)\}$

$$\hat{S}_7 = \{(1,3),(2,9),(3,13),(4,19),(5,21),(6,27),(7,31),(8,37),(11,42),(12,46),(13,52),(16,57),(20,59),(23,64)\}$$
$$S_7 = \{(0,0),(1,6),(2,10),(3,16),(4,19),(5,24),(6,28),(7,34),(8,37),(10,39),$$
$$(11,43),(12,49),(13,52),(15,54),(16,57),(20,59),(22,61),(23,64)\}$$

$f_7(15) = 54.$

Berechnung einer optimierende Auswahl $x_1^*, ..., x_n^*$:

$S_0 = \{(\boldsymbol{0,0})\}$

$k = 1,\ W = 2,\ P = 10, \qquad x_1^* = 1, \qquad W = 0,\ P = 0$

$\hat{S}_1 = \{(\boldsymbol{2,10})\} \qquad S_1 = \{(0,0),(\boldsymbol{2,10})\}$

$k = 2,\ W = 10,\ P = 30, \qquad x_2^* = 1, \qquad W = 2,\ P = 10$

$\hat{S}_2 = \{(3,5),(\boldsymbol{5,15})\} \qquad S_2 = \{(0,0),(2,10),(\boldsymbol{5,15})\}$

$k = 3,\ W = 10,\ P = 30, \qquad x_3^* = 1, \qquad W = 5,\ P = 15$

$\hat{S}_3 = \{(5,15),(7,25),(\boldsymbol{10,30})\}$
$S_3 = \{(0,0),(2,10),(5,15),(7,25),(\boldsymbol{10,30})\}$

$k = 4,\ W = 10,\ P = 30, \qquad x_4^* = 0$

$\hat{S}_4 = \{(7,7),(9,17),(12,22),(14,32),(17,37)\}$
$S_4 = \{(0,0),(2,10),(5,15),(7,25),(\boldsymbol{10,30}),(14,32),(17,37)\}$

$k = 5,\ W = 11,\ P = 36, \qquad x_5^* = 1, \qquad W = 10,\ P = 30$

$\hat{S}_5 = \{(1,6),(3,16),(6,21),(8,31),(\boldsymbol{11,36}),(15,38),(18,43)\}$
$S_5 = \{(0,0),(1,6),(2,10),(3,16),(6,21),(7,25),(8,31),(\boldsymbol{11,36}),(15,38),(18,43)\}$

$k = 6,\ W = 15,\ P = 54, \qquad x_6^* = 1, \qquad W = 11,\ P = 36$

$\hat{S}_6 = \{(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(\boldsymbol{15,54}),(19,56),(22,61)\}$

$S_6 = \{(0,0),(1,6),(2,10),(3,16),(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(\boldsymbol{15,54}),(19,56),(22,61$

$k = 7,\ W = 15,\ P = 54, \qquad x_7^* = 0$

$$\hat{S}_7 = \{(1,3),(2,9),(3,13),(4,19),(5,21),(6,27),(7,31),(8,37),(11,42),(12,46),(13,52),(16,57),(20,59),(23,64)\}$$
$$S_7 = \{(0,0),(1,6),(2,10),(3,16),(4,19),(5,24),(6,28),(7,34),(8,37),(10,39),$$
$$(11,43),(12,49),(13,52),(\mathbf{15,54}),(16,57),(20,59),(22,61),(23,64)\}$$

Es gilt:

Ist $I = (A, M)$ eine Instanz des 0/1-Rucksackproblems (als Maximierungsproblem) mit $size(I) = n \cdot \lceil \log(p_{max} + 1) \rceil$, dann liefert das beschriebene Verfahren eine optimale Lösung mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n \cdot \sum_{i=1}^{n} p_i)$; hierbei ist p_i der Profit des *i*-ten Eingabeelements. Das Verfahren hat also exponentielle Laufzeit in der Größe der Eingabe.

Bis heute ist kein Verfahren bekannt, das in polynomieller Laufzeit eine optimale Lösung für das 0/1-Rucksackproblem liefert. Wahrscheinlich wird es auch kein Verfahren geben. In einigen Spezialfällen lassen sich jedoch schnelle Lösungsverfahren angeben. Das folgende Beispiel aus der Kryptologie beschränkt die Eingabeinstanzen auf „schnell wachsende“ (super increasing) Folgen und einfache Eingabewerte:

Das 0/1-Rucksackproblem mit schnell wachsenden (super increasing) ganzzahligen Eingabewerten als Entscheidungsproblem

Instanz: $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von *n* natürlichen Zahlen mit der Eigenschaft $a_j > \sum_{i=1}^{j-1} a_i$ (super increasing) und $M \in \mathbf{N}$.

$size(I) = n \cdot \lceil \log_2(a_n + 1) \rceil$; diese Größe ist proportional zur Anzahl der Bits, die zur Darstellung von *I* benötigt werden.

Lösung: Entscheidung „ja“, falls es eine Folge $x_1, ..., x_n$ von Zahlen mit $x_i = 0$ oder $x_i = 1$ für $i = 1, ..., n$ gibt, so daß $\sum_{i=1}^{n} x_i \cdot a_i = M$ gilt,

Entscheidung „nein“ sonst.

Die folgenden (Pseudocode-) Anweisungen nach der Greedy-Methode ermitteln eine Folge $x_1, ..., x_n$ mit der gewünschten Eigenschaft, falls es sie gibt:

```
TYPE Entscheidungstyp = (ja, nein);
VAR  Entscheidung : Entscheidungstyp;

summe := 0;
FOR i := n DOWNTO 1 DO
  IF a_i + summe ≤ M
  THEN BEGIN
         summe := summe + a_i;
         x_i := 1
       END
  ELSE x_i := 0;
IF summe = M THEN Entscheidung = ja
             ELSE Entscheidung = nein;
```

Beispiel: $I = (\{a_1, ..., a_7\}, 234)$ mit $a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 11, a_5 = 32, a_6 = 87, a_7 = 141$.

a_i + summe $\leq M$	x_i
a_7 + 0 = 141 < 234	$x_7 = 1$
summe = 141	
a_6 + 141 = 87 + 141 = 228 < 234	$x_6 = 1$
summe = 228	
a_5 + 228 = 32 + 228 = 260 > 234	$x_5 = 0$
summe = 228	
a_4 + 228 = 11 + 228 = 239 > 234	$x_4 = 0$
summe = 228	
a_3 + 228 = 5 + 228 = 233 < 234	$x_3 = 1$
summe = 233	
a_2 + 233 = 2 + 233 = 235 > 234	$x_2 = 0$
summe = 233	
a_1 + 233 = 1 + 233 = 234	$x_1 = 1$
summe = 234	
Entscheidung „ja“	

Es gilt:

Ist $I = (A, M)$ eine Instanz des 0/1-Rucksackproblems mit schnell wachsenden Eingabewerten als Entscheidungsproblem mit $size(I) = k$, dann liefert das beschriebene Verfahren eine Entscheidung mit einer (worst-case-) Zeitkomplexität der Ordnung $O(k)$.

Die Methode der Dynamischen Programmierung führt nicht in jedem Fall auf Algorithmen mit exponentiellem Laufzeitverhalten, wie folgendes Beispiel zeigt.

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen zwischen allen Knotenpaaren

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}$ gibt jeder Kante $e \in E$ ein Gewicht mit der Eigenschaft, daß keine Zyklen mit negativem Gewicht entstehen.

$size(G) = n$.

Lösung: Die minimalen Gewichte $d_{i,j}$ der Wege vom Knoten v_i zum Knoten v_j des Graphs für $i = 1, ..., n$ und $j = 1, ..., n$.

Bemerkung: Es sind jetzt auch Instanzen mit Gewichtungen zugelassen, die negativ sind. Jedoch dürfen jedoch keine Zyklen mit negativem Gewicht vorkommen.

Der folgende Algorithmus realisiert das Prinzip der Dynamischen Programmierung. Dazu wird das Problem, nämlich für jedes Knotenpaar einen Weg mit minimalem Gewicht zu bestimmen, wobei alle Knoten des Graphs als Zwischenknoten zugelassen sind, in kleinere Probleme aufgeteilt. Dazu wird $A^k(i, j)$ für $k = 1, ..., n$ als das minimale Gewicht eines Wegs vom Knoten v_i zum Knoten v_j definiert, der nur Zwischenknoten aus $\{v_1, ..., v_k\}$ enthält, also keine Zwischenknoten mit einer Nummer, die größer als k ist. Gesucht ist $A^n(i, j) = d_{i,j}$. Das Problem der Bestimmung von $A^n(i, j)$ wird aufgeteilt in die Bestimmung der kleineren Probleme der Bestimmung von $A^k(i, j)$ für $k = 1, ..., n$ in dieser Reihenfolge.

Ein Weg mit minimalem Gewicht vom Knoten v_i zum Knoten v_j definiert, der nur Zwischenknoten aus $\{v_1, ..., v_k\}$ enthält, geht entweder nicht durch den Knoten v_k (dann ist $A^k(i, j) = A^{k-1}(i, j)$) oder er führt durch v_k. In diesem Fall ist $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$. Insgesamt ist

$A^k(i, j) = \min\{A^{k-1}(i, j),\ A^{k-1}(i, k) + A^{k-1}(k, j)\}$ für $k = 1, ..., n$
mit $A^0(i, j) = w((v_i, v_j))$.

Zur algorithmischen Behandlung wird der Graph $G = (V, E, w)$ in Form seiner Adjazenzmatrix $A(G)$ gespeichert.

$$A(G) = A_{(n,n)} = \lfloor a_{i,j} \rfloor_{(n,n)} \text{ mit } a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}.$$

Es werden wieder folgende Datentypen verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden       }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      feld          = ARRAY [knotenbereich] OF REAL;
```

Die folgende Funktion ermittelt das Minimum zweier REAL-Zahlen:

```
FUNCTION min (a, b: REAL):REAL;
```

Algorithmus zur Lösung des Problems der Wege mit minimalem Gewicht in gerichteten Graphen zwischen allen Knotenpaaren:

Eingabe: $G = (V, E, w)$ ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{R}$ gibt jeder Kante $e \in E$ ein Gewicht, wobei keine Zyklen mit negativem Gewicht vorkommen dürfen.

```
VAR A       : matrix;        { Adjazenzmatrix }
    i       : knotenbereich;
    j       : knotenbereich;
    distanz : matrix;

FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i, v_j) ∈ E THEN A[i, j] := w((v_i, v_j))
                      ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `alle_minimalen_wege (A, distanz);`

Ausgabe: $d_{i,j}$ = `distanz[i,j]` für i = 1, ..., n und j = 1, ..., n.

```
PROCEDURE alle_minimalen_wege (A          : matrix;
```

```
                                VAR distanz : matrix;)

  { die Prozedur ermittelt das minimale Gewicht eines Wegs
    zwischen je zwei Knoten v_i und v_j (in distanz[i, j])in
    dem durch die Adjazenzmatrix A gegebenen Graphen.
    Der Graph darf keine negativen Zyklen enthalten.
    Außerdem wird A[i, i] = 0 vorausgesetzt.                    }

  VAR idx         : knoten_bereich;
      jdx         : knoten_bereich;
      kdx         : knoten_bereich;

   BEGIN { alle_minimalen_wege }

     { Kostenwegematrix initialisieren }
     FOR idx := 1 TO n DO
      FOR jdx := 1 TO n DO
        distanz[idx, jdx] := A[idx, jdx];

     FOR kdx := 1 TO n DO
      FOR idx := 1 TO n DO
       FOR jdx := 1 TO n DO
        distanz[idx, jdx]
              := min(distanz[idx, jdx],
                     distanz [idx, kdx] + distanz [kdx, jdx]);

   END { alle_minimalen_wege };
```

Ist $G = (V, E, w)$ eine Instanz des Problems der Wege mit minimalem Gewicht zwischen allen Knotenpaaren in gerichteten Graphen mit $size(G) = n$, dann liefert das beschriebene Verfahren mit der Prozedur `alle_minimalen_wege` im Feld `distanz` die minimalen Gewichte $d_{i,j}$ der Wege vom Knoten v_i zu allen Knoten v_j des Graphs für $i = 1, ..., n$ und $j = 1, ..., n$. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n^3)$.

2.4 Branch and Bound

Die Branch and Bound-Methode liefert in vielen praktischen Fällen eine sehr effiziente Lösungsmethode für Probleme, deren Lösungsraum mindestens exponentiell großen Umfang aufweist. Sie wird wieder am Problem des Handlungsreisenden auf Graphen (minimum traveling salesperson problem), siehe Kapitel 2.3, erläutert:

Eine Eingabeinstanz $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht; die Größe einer Instanz ist $size(G) = n$. Gesucht wird eine Eine Tour $T^* \in \text{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht, und $m(G, T^*)$.

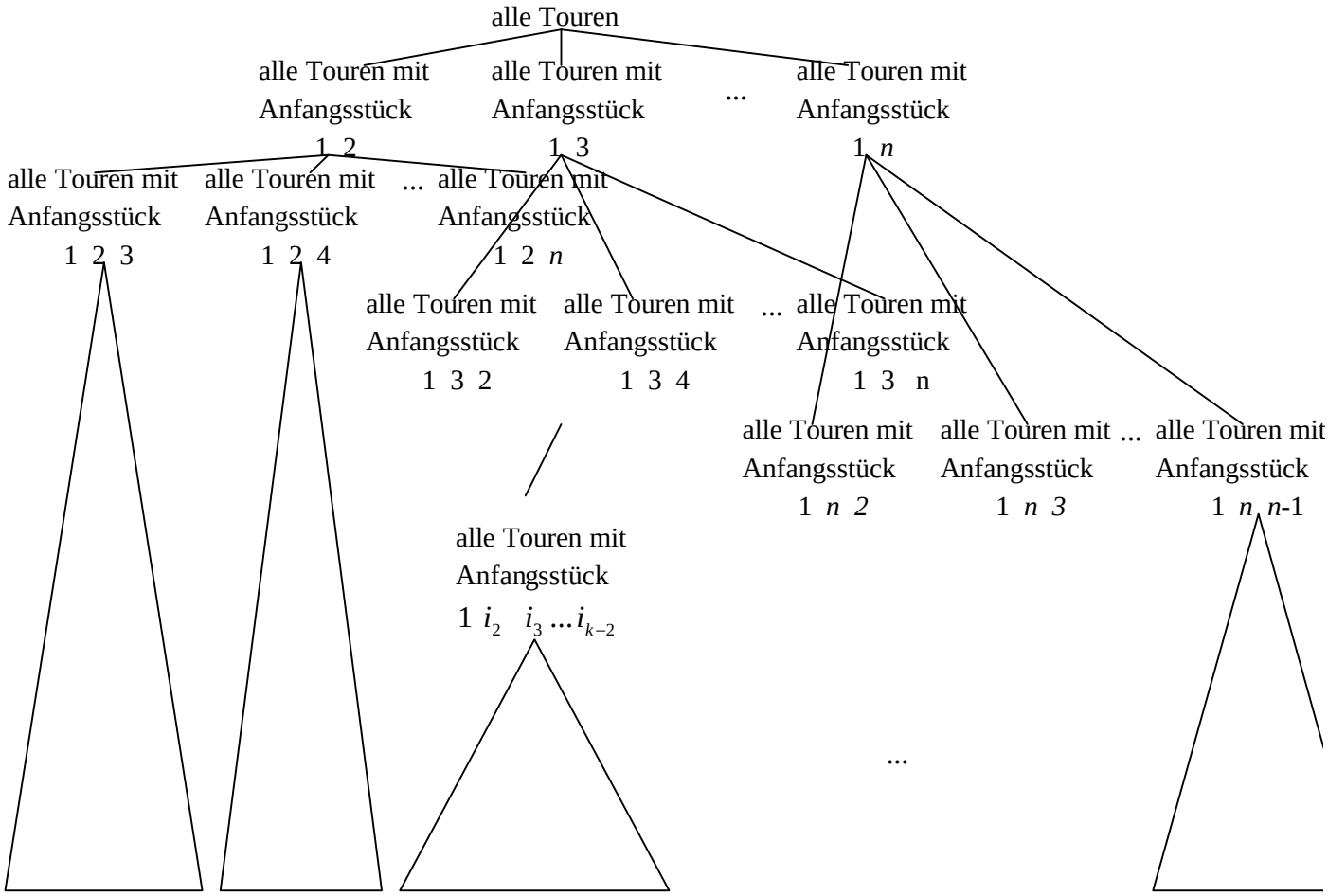
Im folgenden wird wieder angenommen, daß eine Tour bei v_1 beginnt und endet. Außerdem wird angenommen, daß der Graph **vollständig** ist, d.h. daß $w((v_i, v_j))$ für jedes $v_i \in V$ und $v_j \in V$ definiert ist (eventuell ist $w((v_i, v_j)) = \infty$).

Im folgenden wird (zur Vereinfachung der Darstellung) die Knotenmenge $V = \{v_1, ..., v_n\}$ mit der Menge der Zahlen $\{1, ..., n\}$ gleichgesetzt (dem Knoten v_i entspricht die Zahl *i*). Eine Tour durch *G* läßt sich dann als Zahlenfolge

$\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1 \rangle$

darstellen, wobei alle i_j paarweise verschieden (und verschieden von 1) sind. Alle Touren erhält man, wenn man für i_j in $\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1 \rangle$ alle $(n-1)!$ Permutationen der Zahlen 2, ..., *n* einsetzt. Manche dieser Touren haben eventuell das Gewicht ∞.

Alle Touren (Zahlenfolgen der beschriebenen Art) lassen sich als Blätter eines Baums darstellen:



Alle Touren (Zahlenfolgen der beschriebenen Art) werden systematisch erzeugt (siehe unten). Dabei wird eine obere Abschätzung für das Gewicht einer optimalen Tour (Tour mit minimalem Gewicht) in der globalen Variablen `bound` mitgeführt (Anfangswert `bound` = ∞). Wird ein Anfangsstück einer Tour erzeugt, deren Gewicht größer als der Wert in der Variablen `bound` ist, dann müssen alle Touren, die mit diesem Anfangsstück beginnen, nicht weiter betrachtet werden. Es wird also ein ganzer Teilbaum aus dem Baum aller Touren abgeschnitten. Ist schließlich eine Tour gefunden, deren Gewicht kleiner oder gleich dem Wert in der Variablen `bound` ist, so erhält die Variable `bound` als neuen Wert dieses Gewicht, und die Tour wird als temporär optimale Tour vermerkt. Eine Variable `opttour` nimmt dabei die Knotennummern einer temporär optimalen Tour auf; die Variable `teilfeld` dient der Aufnahme des Anfangsstücks einer Tour.

Ist bereits ein Anfangsstück $\langle 1\ i_1\ i_2 \ldots i_{k-1} \rangle$ einer Tour erzeugt, so sind die Zahlen $1, i_1, i_2, \ldots, i_{k-1}$ alle paarweise verschieden. Eine weitere Zahl *i* kann in die Folge aufgenommen werden, wenn

(1) *i* unter den Zahlen $1, i_1, i_2, \ldots, i_{k-1}$ nicht vorkommt und
(2) $w((v_{i_{k-1}}, v_i)) < \infty$ ist und
(3) das Gewicht der neu entstehenden Teiltour $\langle 1, i_1, i_2, \ldots, i_{k-1}\ i \rangle$ kleiner oder gleich dem Wert in der Variablen `bound` ist.

Treffen (1) oder (2) nicht zu, kann *i* nicht als Fortsetzung der Teiltour $1, i_1, i_2, \ldots, i_{k-1}$ genommen werden, denn es entsteht keine Tour. Trifft (3) nicht zu (aber (1) und (2)), dann brauchen alle Touren, die mit dem Anfangsstück $\langle 1, i_1, i_2, \ldots, i_{k-1}\ i \rangle$ beginnen, nicht weiter berücksichtigt zu werden.

Algorithmus zur Lösung des Problems des Handlungsreisenden nach der Branch-and-bound-Methode:

Es werden wieder die bereits bekannten Deklarationen (ergänzt um neue Deklarationen) verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden      }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      tourfeld      = ARRAY [1..(n+1)] OF INTEGER;
```

Die Knotennummern einer optimalen Tour werden im Feld `opttour` abgelegt.

Eingabe: $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

```
VAR A      : matrix;        { Adjazenzmatrix }
    i      : knotenbereich;
    j      : knotenbereich;
    opttour : tourfeld;
    bound  : REAL;
```

```
FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i,v_j) ∈ E THEN A[i, j] := w((v_i,v_j))
                     ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `salesman_bab (A, opttour, bound).`

Ausgabe: `bound` gibt das minimale Gewicht einer Tour durch G an, die beim Knoten v_1 beginnt und endet; `opttour` enthält die Knotennummern einer optimalen Tour.

```
PROCEDURE salesman_bab
                (A           : matrix;
                 VAR opttour : tourfeld; { optimale Tour        }
                 VAR bound   : REAL      { Länge der Tour       }
                 );

  VAR i : INTEGER;

  PROCEDURE  bab_g
                (teilfeld : tourfeld;
                                        { zu ergänzendes Anfangsstück
                                          einer Tour               }
                 gewicht  : REAL;     { Gewicht des Anfangsstücks }
                 position : INTEGER { Position, an der zu
                                          ergänzen ist             }
                );

    VAR i  : INTEGER;
        j  : INTEGER;
        ok : BOOLEAN;
        w  : REAL;

    BEGIN {bab_g }

      IF position = n + 1
      THEN BEGIN
             w := A[teilfeld[n], 1];
             IF (w < unendlich)
                AND
                (gewicht + w < bound)
             THEN BEGIN
                    teilfeld[position] := 1;
                    bound   := gewicht + w;
                    opttour := teilfeld;
```

```
                END;
          END
      ELSE BEGIN
             FOR i := 2 TO n DO
               BEGIN
                 w  := A[teilfeld[position - 1], i];
                 ok := TRUE;
                 { Bedingung (2):}
                 FOR j := 2 TO position - 1 DO
                   IF teilfeld[j] = i
                   THEN BEGIN
                          ok := FALSE;
                          Break;
                        END;
                 IF ok
                    AND
                    (w < unendlich)
                    AND
                    (gewicht + w < bound)
                 THEN BEGIN
                        teilfeld[position] := i;
                        bab_g (teilfeld, gewicht + w,
                                                 position + 1);
                      END;
               END;
          END;

   END   {bab_g };

BEGIN { salesman_bab }

  FOR i := 2 TO n + 1 DO
   opttour[i]  := 0;
  opttour[1] := 1;
  bound   := unendlich;

  bab_g(opttour, 0, 2);

END   {salesman_bab };
```

Das Verfahren erzeugt u.U. alle $(n-1)!$ Folgen $\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1\rangle$, bis es eine optimale Tour gefunden hat. In der Praxis werden jedoch schnell Folgen mit Anfangsstücken, die ein zu großes Gewicht aufweisen, ausgeschlossen. Die (worst-case-) Zeitkomplexität des Verfahrens bleibt jedoch mindestens exponentiell in der Anzahl der Knoten des Graphen in der Eingabeinstanz.

3 Praktische Berechenbarkeit

In den vorhergehenden Kapiteln wurde eine Reihe von Algorithmen behandelt, deren Laufzeitverhalten exponentiell oder polynomiell in der Größe der Eingabe ist. In der Praxis gilt ein Algorithmus mit exponentiellem Laufzeitverhalten als **schwer durchführbar (intractable)**, ein Algorithmus mit polynomiellem Laufzeitverhalten als **leicht durchführbar (tractable)**. Natürlich kann dabei ein Algorithmus mit exponentiellem Laufzeitverhalten für einige Eingaben oder Eingabentypen schnell ablaufen. Trotzdem bleibt die Ursache zu untersuchen, die dazu führt, daß manche Probleme „leicht lösbar“ sind (d.h. einen Algorithmus mit polynomiellem Laufzeitverhalten besitzen), während für andere Probleme bisher nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten bekannt sind.

Wie in Kapitel 1.3 bereits erläutert bedarf es dabei einer sorgfältigen Festlegung der Problemgröße, insbesondere dann, wenn numerische Werte der Eingabeinstanz einen Einfluß auf das Laufzeitverhalten haben. Das folgende Beispiel erläutert diese Aussage noch einmal.

Das Partitionenproblem mit ganzzahligen Eingabewerten als Entscheidungsproblem

Instanz: $I = \{a_1, ..., a_n\}$

I ist eine Menge von *n* natürlichen Zahlen.

$size(I) = n \cdot \log_2(B)$ mit $B = \sum_{i=1}^{n} a_i$; diese Größe ist proportional zur Anzahl der Bits, die zur Darstellung von *I* benötigt werden.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Offensichtlich lautet bei einer Instanz $I = \{a_1, ..., a_n\}$ mit ungeradem $B = \sum_{i=1}^{n} a_i$ die Entscheidung „nein“. Daher kann *B* als gerade vorausgesetzt werden.

Der folgende Algorithmus berechnet für seine Entscheidung den Boolesche Ausdruck $T[i, j]$, der durch

$T[i, j] =$ TRUE genau dann, wenn es eine Teilmenge von $\{a_1, ..., a_i\}$ gibt, deren Elemente sich auf genau *j* aufsummieren

definiert ist.

Eine Instanz $I = \{a_1, ..., a_n\}$ führt also genau dann zur Entscheidung „ja“, wenn $T[n, B/2] = \text{TRUE}$ ist.

Man kann sich die Werte $T[i, j]$ in einer Tabelle mit *n* Zeilen und *B*/2 + 1 Spalten (numeriert von 0 bis *B*/2) angeordnet vorstellen. Die Werte der Zeile 1 können direkt aus der Definition von *T* eingetragen werden:

$T[1,0] = \text{TRUE}$, $T[1, a_1] = \text{TRUE}$, $T[1, j] = \text{FALSE}$ für $j \neq 0$ und $j \neq a_1$.

Für die übrigen Zeilen der Tabelle gilt die Rekursion

> $T[i, j] = \text{TRUE}$ genau dann, wenn entweder $T[i-1, j] = \text{TRUE}$ ist (hier wird der Eintrag in der vorherigen Zeile und derselben Spalte benötigt) oder wenn $a_i \leq j$ und $T[i-1, j-a_i] = \text{TRUE}$ ist (hier wird der Eintrag in der vorhergehenden Zeile und einer Spalte mit kleinerem Index benötigt).

Beispiel: $I = \{1, 9, 5, 3, 8\}$, $B = 26$

$T[i, j]$:

j / *i*	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2	T	T								T	T			
3	T	T				T	T			T	T			
4	T	T		T	T	T	T		T	T	T		T	T
5	T	T		T	T	T	T		T	T	T	T	T	T

T = TRUE, alle leeren Einträge entsprechen dem Wert FALSE.

Die Zerlegung von *I* lautet $I = \{1, 9, 3\} \cup \{5, 8\}$.

Für eine Implementierung bietet sich die Methode der Dynamischen Programmierung an. Dabei wird die Tabelle $T[i, j]$ zeilenweise für $i = 1, ..., n$ aufgebaut, wobei es genügt, sich immer nur zwei Zeilen der Tabelle zu merken, nämlich diejenigen, die die Werte $T[i-1, j]$ bzw. $T[i, j]$ enthalten. Man kennt jedoch selbst bei festem *n* von vornherein nicht die Anzahl $B/2+1$ der Spalten, d.h. die Länge einer Zeile der Tabelle. Daher wird in der folgenden Implementierung der Speicherplatz für die beiden jeweils benötigten Zeilen der Tabelle erst nach Lesen der Eingabeinstanz $I = \{a_1, ..., a_n\}$ dynamisch allokiert.

Algorithmus zur Lösung des Partitionenproblems:

Eingabe: $I = \{a_1, ..., a_n\}$ eine Menge von *n* natürlichen Zahlen.

```
TYPE feld_typ         = ARRAY[1..n] OF INTEGER;
     Entscheidungstyp = (ja, nein);

VAR a : feld_typ;
    I : INTEGER;

FOR i := 1 TO n DO a[i] := a_i;
```

Verfahren: Aufruf der Prozedur `partition (a, n, Entscheidung)`

Ausgabe: `Entscheidung = ja`, falls es falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
`Entscheidung = nein` sonst.

```
PROCEDURE partition (a                : feld_typ;
                     n                : INTEGER;
                     VAR Entscheidung : Entscheidungstyp);

CONST max_INTEGER = ...;

TYPE Zeilen_typ = ARRAY [0 .. max_INTEGER] OF BOOLEAN;

VAR i             : INTEGER;
    j             : INTEGER;
    B             : INTEGER;
    Zeile_i_1_ptr : Pointer;  { Zeiger auf die erste Zeile  }
    Zeile_i_ptr   : Pointer;  { Zeiger auf die zweite Zeile }
    dimension     : INTEGER;
    ptr           : Pointer;

BEGIN { partition }
  B := 0;
  FOR i := 1 TO n DO B := B + a[i];
  IF (B MOD 2) = 1
  THEN Entscheidung := nein
  ELSE BEGIN
         { Speicherplatz für die beiden Zeilen der Tabelle T
           allokieren:                                       }
         dimension := B DIV 2 + 1;
```

```
          GetMem (Zeile_i_1_ptr, dimension);
          GetMem (Zeile_i_ptr, dimension);
          dimension := dimension – 1;

          { Zeile 1 mit Werten belegen:                          }
          FOR j := 0 TO dimension DO
             Zeilen_typ(Zeile_i_1_ptr^)[j] := FALSE;
          Zeilen_typ(Zeile_i_1_ptr^)[0]    := TRUE;
          Zeilen_typ(Zeile_i_1_ptr^)[a[1]] := TRUE;

          { Sukzessive die zweite Zeile aufbauen:                }
          FOR i := 2 TO n DO
             BEGIN
               FOR j := 0 TO dimension DO
                  IF Zeilen_typ(Zeile_i_1_ptr^)[j]
                     OR
                     (a[i] <= j)
                       AND Zeilen_typ(Zeile_i_1_ptr^)[j - a[i]]
                  THEN Zeilen_typ(Zeile_i_ptr^)[j] := TRUE
                  ELSE Zeilen_typ(Zeile_i_ptr^)[j] := FALSE;
               { zweite Zeile zur ersten machen:                 }
               ptr           := Zeile_i_1_ptr;
               Zeile_i_1_ptr := Zeile_i_ptr;
               Zeile_i_ptr   := ptr;
             END;

          IF Zeilen_typ(Zeile_i_1_ptr^)[dimension]
          THEN Entscheidung = ja
          ELSE Entscheidung = nein;

          dimension := dimension + 1;
          FreeMem (Zeile_i_1_ptr, dimension);
          FreeMem (Zeile_i_ptr, dimension);

        END
END    { partition };
```

Offensichtlich ist ist die Berechnung aller Tabelleneinträge von $T[i, j]$ bzw. die Berechnung aller Werte der beiden Zeilen über alle Schleifendurchläufe der Prozedur `partition` von der Ordnung $O(n \cdot B)$. Die Eingabe $I = \{a_1, ..., a_n\}$ hat die Größe $size(I) = n \cdot \log_2(B)$ mit $B = \sum_{i=1}^{n} a_i$, d.h. die Laufzeit von `partition` ist exponentiell in der Größe der Eingabe.

Beschränkt man jedoch die Größen der Zahlen in der Eingabeinstanz $I = \{a_1, ..., a_n\}$, d.h. gilt $a_i \leq c$ für $i = 1, ..., n$ mit einer Konstanten c, so hat obiges Verfahren polynomielles Laufzeitverhalten. Man spricht in diesem Fall von einem pseudo-polynomiellem Algorithmus:

Ein Algorithmus, der numerische Eingaben verarbeitet und dessen Laufzeit von den Größen der Eingaben abhängt, heißt **pseudo-polynomiell**, wenn er bei Beschränkung der Eingabegrößen durch eine Konstante polynomielles Laufzeitverhalten aufweist.

Für das Partitionenproblem gibt es also einen pseudo-polynomiellen Algorithmus, während für andere Probleme, wie z.B. das Problem des Handlungsreisenden kein pseudo-polynomieller Algorithmus bekannt ist.

Im folgenden werden zunächst hauptsächlich Entscheidungsprobleme behandelt (und nicht Optimierungsprobleme), da diese einfach zu formulieren sind (einfache „ja"/„nein"-Entscheidungen). Zudem befassen sich die Grundlagen der Komplexitätstheorie mit dem Thema Berechenbarkeit/Entscheidbarkeit mit dem Berechnungsmodell Turing-Maschine. Dieses Modell behandelt primär Entscheidungsprobleme.

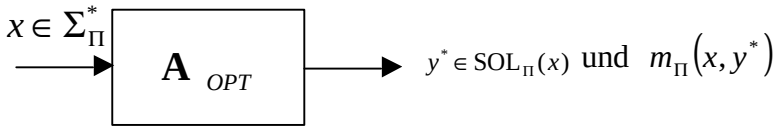
3.1 Zusammenhang zwischen Optimierungsproblemen und zugehörigen Entscheidungsproblemen

Es sei Π ein Optimierungsproblem mit einer reellwertigen Zielfunktion:

Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ eine Menge zulässiger Lösungen zuordnet
3. Spezifikation einer Zielfunktion m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den Wert einer zulässigen Lösung, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$)
bzw.
$m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

Ein Lösungsalgorithmus $\mathbf{A}_{OPT}$ für Π hat die Form



Die von $\mathbf{A}_{OPT}$ bei Eingabe von $x \in \Sigma_\Pi^*$ ermittelte Lösung ist $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$.

Das zu Π **zugehörige Entscheidungsproblem** Π_{ENT} wird definiert durch

Instanz: $[x, K]$
mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{R}$

Lösung: Entscheidung „ja“, falls für den Wert $m_\Pi^*(x)$ einer optimalen Lösung
$m_\Pi^*(x) \geq K$ bei einem Maximierungsproblem
(d.h. $goal_\Pi = \max$)
bzw.
$m_\Pi^*(x) \leq K$ bei einem Minimierungsproblem
(d.h. $goal_\Pi = \min$)
gilt,
Entscheidung „nein“, sonst.

Hierbei ist zu beachten, daß beim Entscheidungsproblem weder nach einer optimalen Lösung noch nach dem Wert $m_\Pi^*(x)$ der Zielfunktion bei einer optimalen Lösung gefragt wird.

Aus der Kenntnis eines Algorithmus $\mathbf{A}_{OPT}$ für ein Optimierungsproblem läßt sich leicht ein Algorithmus $\mathbf{A}_{ENT}$ für das zugehörige Entscheidungsproblem konstruieren, der im wesentlichen dieselbe Komplexität besitzt:

Eingabe: $[x, K]$
mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{R}$

Verfahren: Man berechne $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$ und vergleiche das Resultat $m_\Pi(x, y^*) = m_\Pi^*(x)$ mit K:

Ausgabe: $\mathbf{A}_{ENT}([x, K]) = \mathtt{ja}$, falls $m_\Pi^*(x) \geq K$ bei einem Maximierungsproblem ist
bzw.
$\mathbf{A}_{ENT}([x, K]) = \mathtt{ja}$, falls $m_\Pi^*(x) \leq K$ bei einem Minimierungsproblem ist,

$\mathbf{A}_{ENT}([x,K]) = \texttt{nein}$ sonst.

Daher ist das zu einem Optimierungsproblem Π zugehörige Entscheidungsproblem Π_{ENT} im wesentlichen *nicht schwieriger zu lösen* als das Optimierungsproblem.

Falls man andererseits bereits weiß, daß das Entscheidungsproblem Π_{ENT} immer nur „schwer lösbar“ ist, z.B. beweisbar nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten besitzt, dann ist das Optimierungsproblem ebenfalls nur „schwer lösbar“.

In einigen Fällen läßt sich auch die „umgekehrte“ Argumentationsrichtung zeigen: Aus einem Algorithmus $\mathbf{A}_{ENT}$ für das zu einem Optimierungsproblem zugehörige Entscheidungsproblem läßt sich ein Algorithmus $\mathbf{A}_{OPT}$ für das Optimierungsproblem konstruieren, dessen Laufzeitverhalten im wesentlichen dieselbe Komplexität aufweist. Insbesondere ist man dabei an Optimierungsproblemen interessiert, für die gilt: Hat $\mathbf{A}_{ENT}$ zur Lösung des zu einem Optimierungsproblem zugehörigen Entscheidungsproblem polynomielles Laufzeitverhalten (in der Größe der Eingabe), so hat auch der aus $\mathbf{A}_{ENT}$ konstruierte Algorithmus $\mathbf{A}_{OPT}$ zur Lösung des Optimierungsproblems polynomielles Laufzeitverhalten.

Ein Beispiel ist das Problem des Handlungsreisenden auf Graphen mit natürlichzahligen Gewichten:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimierungsproblem

Instanz:

1. $G = (V, E, w)$
   $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein **natürlichzahliges Gewicht**; es ist $w((i,j)) = \infty$ für $(i,j) \notin E$
   $size(G) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$
2. $\mathrm{SOL}(G) = \left\{T \mid T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle \text{ ist eine Tour durch } G\right\}$
3. für $T \in \mathrm{SOL}(G)$, $T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle$, ist die Zielfunktion definiert durch $m(G,T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$
4. $goal = \min$

Lösung: Eine Tour $T^* \in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G,T^*)$.

Ein Algorithmus, der bei Eingabe einer Instanz $G=(V,E,w)$ eine optimale Lösung erzeugt, werde mit $\mathbf{A}_{TSPOPT}$ bezeichnet. Die von $\mathbf{A}_{TSPOPT}$ bei Eingabe von G ermittelte Tour mit minimalen Kosten sei T^*, die ermittelten minimalen Kosten $m^*(G)$:
$\mathbf{A}_{TSPOPT}(G)=(T^*,m^*(G))$.

Das zugehörige Entscheidungsproblem lautet:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Entscheidungsproblem

Instanz: $[G,K]$
$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; es ist $w((i,j))=\infty$ für $(i,j) \notin E$;
$K \in \mathbf{N}$.
$size([G,K])=k=n \cdot B$ mit $B=\max\{\lceil \log(w(e)) \rceil \mid e \in E\}$.

Lösung: Entscheidung „ja“, falls es eine Tour durch G gibt mit minimalen Kosten $m^*(G) \le K$ gibt,
Entscheidung „nein“, sonst.

Ein Algorithmus, der bei Eingabe einer Instanz $[G,K]$ eine entsprechende Entscheidung fällt, werde mit $\mathbf{A}_{TSPENT}$ bezeichnet. Die von $\mathbf{A}_{TSPENT}$ bei Eingabe von $[G,K]$ getroffene `ja/nein`-Entscheidung sei $\mathbf{A}_{TSPENT}([G,K])$. Mit Hilfe von $\mathbf{A}_{TSPENT}$ wird ein Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem konstruiert. Dabei wird wiederholt Binärsuche eingesetzt, um zunächst die Kosten einer optimalen Tour zu ermitteln.

Der Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem verwendet eine als Pseudocode formulierte Prozedur `A_TSPOPT`. Der Eingabegraph kann wieder in Form seiner Adjazenzmatrix verarbeitet werden. Da Implementierungsdetails hier nicht weiter betrachtet werden sollen, können wir annehmen, daß es zur Darstellung eines gewichteten Graphen einen geeigneten Datentyp

```
TYPE gewichteter_Graph = ...;
```

und zur Speicherung einer Tour (Knoten und Kanten) in dem Graphen einen Datentyp

```
TYPE tour = ...;
```

gibt. Der Algorithmus $\mathbf{A}_{TSPOPT}$ hat folgendes Aussehen:

Eingabe: $G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; ; es ist $w((i,j))=\infty$ für $(i,j) \notin E$

```
VAR G      : gewichter_Graph;
    opttour : tour;
    opt     : INTEGER;
```

G := $G=(V,E,w)$

Verfahren: Aufruf der Prozedur `A_TSPOPT (G, opt, opttour).`

Ausgabe: `opttour` = T^*, d.h. eine Tour mit minimalen Kosten, `opt` = die ermittelten minimalen Kosten $m^*(G)$.

```
PROCEDURE A_TSPOPT (G          : gewichter_Graph;
                    {  Graph mit natürlichzahligen Kantengewichten }
                    VAR opt    : INTEGER;
                     { Gewicht einer optimalen Tour               }
                    VAR opttour : graph;
                     { ermittelte Tour mit minimalem Gewicht      }

VAR s : INTEGER;


   PROCEDURE TSPOPT (G       : gewichter_graph;
                     VAR opt : INTEGER;)
        { ermittelt im Parameter opt das Gewicht einer
          optimalen Tour in G :                                   }

      VAR min : INTEGER;
          max : INTEGER;
          t   : INTEGER;

   BEGIN { TSPOPT }
     min := 0;
     max := ∑ w(e) ;
            e∈E

     { Binärsuche auf dem Intervall [0..max]: }
     WHILE max - min >= 1 DO
```

```
      BEGIN
        t := ⌈(min + max) / 2⌉;
        IF A_TSPENT([G, t]) = „ja“
        THEN max := t
        ELSE min := t + 1;
      END;

    { t enthält nun das Gewicht einer optimalen Tour in G: }
    opt := t;
  END   { TSPOPT };

BEGIN { A_TSPOPT }
  TSPOPT (G, opt);

  FOR alle e ∈ E DO
    BEGIN
      ersetze in G das Gewicht w(e) der Kante e durch w(e) + 1;
      TSPOPT (G, s);
      IF s > opt
      THEN { es gibt keine optimale Tour in G, die e nicht enthält }
           ersetze in G das Gewicht w(e) der Kante e durch w(e) - 1;
    END;

  { alle Kanten mit nicht erhöhten Gewichten bestimmen eine Tour
    mit minimalen Kosten                                          }
  tour := Menge der Kanten mit nicht erhöhten Gewichten
          und zugehörige Knoten;
END   { A_TSPOPT };
```

Für einen Graphen G ist $size(G) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$. Es sei $m = \sum_{e \in E} w(e)$. Dann sind in obigem Verfahren höchstens $\lceil \log_2(m) \rceil$ viele Aufrufe des Entscheidungsverfahrens $\mathbf{A}_{TSPENT}([G, t])$ erforderlich (Binärsuche). Es gilt

$$\lceil \log_2(m) \rceil \le \log_2(m) + 1 \le \sum_{e \in E} \log_2(w(e)) + 1 \le \sum_{e \in E} \lceil \log_2(w(e)) \rceil + 1 \le n^2 \cdot B + 1 \in O(k^2),$$

d.h. $\lceil \log_2(m) \rceil \in O(k^2)$.

Falls man also weiß, daß $\mathbf{A}_{TSPENT}$ ein in der Größe der Eingabe polynomiell zeitbeschränkter Algorithmus ist, ist das gesamte Verfahren zur Bestimmung einer optimalen Lösung polynomiell zeitbeschränkt.

Falls man umgekehrt weiß, daß es beweisbar keinen schnellen Algorithmus zur Lösung des Problems des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimie-

rungsproblem gibt, gibt es einen solchen auch nicht für das zugehörige Entscheidungsproblem.

3.2 Komplexitätsklassen

In den bisher behandelten Beispielen werden Eingabeinstanzen für ein Problem über jeweils einem „geeigneten“ Alphabet formuliert. Beispielsweise kann man als Alphabet für das Handlungsreisenden-Minimierungsproblem Π das Alphabet

$\Sigma_\Pi = \{0, 1, (,)\} \cup$ Kommazeichen

wählen. Eine Eingabeinstanz $G = (V, E, w))$ besteht ja aus rationalen Zahlen (die jeweils als Paare natürlicher Zahlen, hier kodiert im Binärformat geschrieben werden können) für die Knotennummern, Paare rationaler Zahlen für die Kanten mit ihren Gewichten und trennenden Klammern und dem Kommazeichen. Entsprechend kann eine Instanz $[(V, E, w)), K]$ für das zugehörige Entscheidungsproblem Π_{ENT} ebenfalls mit Hilfe des Alphabets Σ_Π formuliert werden. $L_{\Pi_{ENT}}$ besteht in diesem Fall aus den Kodierungen aller Instanzen $[(V, E, w)), K]$, für die es eine Tour mit Gewicht $\leq K$ gibt. Ist $Code[(V, E, w)), K]$ die Kodierung einer Instanz $[(V, E, w)), K]$, so ist die Problemgröße $size([(V, E, w)), K])$ so definiert, daß sie mit der Anzahl an Zeichen in $Code[(V, E, w)), K]$ übereinstimmt.

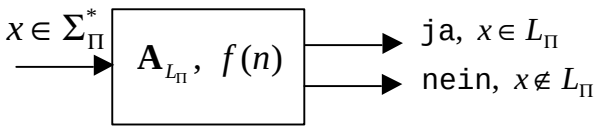
Im folgenden wird daher bei allgemeinen Betrachtungen für *size*(*x*) einer Eingabeinstanz $x \in \Sigma_\Pi^*$ für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ die Anzahl der Zeichen in *x* genommen, d.h. $size(x) = |x|$.

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in TIME(f(n))$, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: ja, falls $x \in L_\Pi$ gilt

nein, falls $x \notin L_\Pi$ gilt.



Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(f(n))$ liegt.

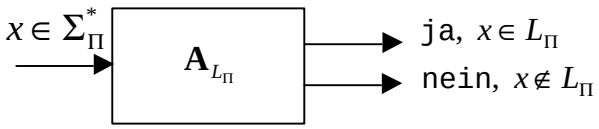
Die Aussage
„das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $TIME(f(n))$“
steht synonym für $L_\Pi \in TIME(f(n))$. In diesem Sinn bezeichnet $TIME(f(n))$ die **Klasse der Entscheidungsprobleme, die in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Eine entsprechende Definition gilt für die Speicherplatzkomplexität:

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in SPACE(f(n))$, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: ja, falls $x \in L_\Pi$ gilt
nein, falls $x \notin L_\Pi$ gilt.



Hierbei werden zur Findung der Entscheidung $\mathbf{A}_{L_\Pi}(x)$ eine Anzahl von Speicherzellen verwendet, die in $O(f(n))$ liegt.

Die Aussage
„das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $SPACE(f(n))$“
steht synonym für $L_\Pi \in SPACE(f(n))$. In diesem Sinn bezeichnet $SPACE(f(n))$ die **Klasse der Entscheidungsprobleme, die mit Speicherplatzkomplexität $O(f(n))$ gelöst werden können.**

Wichtige Komplexitätsklassen sind

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einem Polynom in der Größe der Eingabe ist:

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME(n^k)$$

- die Klasse der Entscheidungsprobleme, die in mit einem Speicherplatzbedarf gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist:

$$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} SPACE\left(n^k\right)$$

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einer Exponentialfunktion in der Größe der Eingabe ist:

  $$\mathbf{EXP} = \bigcup_{k=0}^{\infty} TIME\left(2^{n^k}\right).$$

Es gilt $\mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. Die Frage, ob eine dieser Inklusionen echt ist, d.h. ob $\mathbf{P} \subset \mathbf{PSPACE}$ oder $\mathbf{PSPACE} \subset \mathbf{EXP}$ gilt, ist ein bisher ungelöstes Problem der Komplexitätstheorie. Man weiß jedoch, daß $\mathbf{P} \subset \mathbf{EXP}$ gilt.

3.3 Die Klassen P und NP

Einige Beispiele für Probleme aus der Klasse $\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right)$ wurden in den vorherigen Kapiteln behandelt. Dazu gehört das Problem, festzustellen, ob zwischen zwei Knoten eines gewichteten Graphen ein Pfad mit minimalem Gewicht existiert, das eine vorgegebene Schranke nicht überschreitet.

Entscheidungsprobleme leiten sich häufig von Optimierungsproblemen ab. Leider stellt sich heraus, daß für eine Vielzahl dieser Entscheidungsprobleme keine Lösungsalgorithmen bekannt sind, die polynomielles Laufzeitverhalten aufweisen. Das gilt insbesondere für viele Entscheidungsprobleme, die zu Optimierungsproblemen gehören, die für die Praxis relevant sind (Problem des Handlungsreisenden, Partitionenproblem usw.). Für diese (Optimierungs-) Probleme verfügt man meist über Lösungsalgorithmen, deren Laufzeitverhalten exponentiell in der Größe der Eingabe ist. Derartige Verfahren werden als praktisch nicht durchführbar (intractable) angesehen, obwohl durch den Einsatz immer schnellerer Rechner immer größere Probleme behandelt werden können. Es stellt sich daher die Frage, wieso trotz intensiver Suche nach polynomiellen Lösungsverfahren derartige schnelle Algorithmen (bisher) nicht gefunden wurden. Seit Anfang der 1970‘er Jahre erklärt eine inzwischen gut etablierte Theorie, die **Theorie der NP-Vollständigkeit**, Ursachen dieses Phänomens.

Ein wichtiges Beispiel für Probleme auf der Grenze zwischen **P** und umfassenderen Komplexitätsklasse ist dabei das Problem der Erfüllbarkeit Boolescher Ausdrücke.

Ein **Boolescher Ausdruck** (Boolesche Formel, Formel der Aussagenlogik) ist eine Zeichenkette über dem Alphabet $A = \{\wedge, \vee, \neg, (,), 0, 1, x\}$, der eine Aussage der Aussagenlogik dar-

stellt[3]. Innerhalb eines Booleschen Ausdrucks kommen Variablen vor, die mit dem Zeichen *x* beginnen und von einer Folge von Zeichen aus {0, 1} ergänzt werden. Diese ergänzende 0-1-Folge, die an das Zeichen *x* „gebunden" ist, wird als Indizierung der Variablen interpretiert. Im folgenden werden daher Variablen als indizierte Variablen geschrieben, wobei die indizierende 0-1-Folge als Binärzahl in Dezimaldarstellung angegeben wird, z.B. steht dann anstelle von *x*10111 die indizierte Variable x_{23}. Kommen die Zeichen 0 bzw. 1 innerhalb eines Booleschen Ausdrucks nicht an das Zeichen *x* gebunden vor, so werden sie als Konstanten *FALSCH (FALSE)* bzw. *WAHR (TRUE)* interpretiert. Die Zeichen $\wedge$, $\vee$ bzw. $\neg$ stehen für die üblichen logischen Junktoren *UND, ODER* bzw. *NICHT*. Ein Boolescher Ausdruck wird nur bei korrekter Verwendung der Klammern „(" und „)" als syntaktisch korrekt angesehen: zu jeder sich schließenden Klammer „)" muß es weiter links in der Zeichenkette eine sich öffnende Klammer „(" geben, und die Anzahlen der sich öffnenden und schließenden Klammern müssen gleich sein; außerdem müssen die Junktoren $\wedge$, $\vee$ bzw. $\neg$ korrekt verwendet werden.

Unter einem **Literal** innerhalb eines Booleschen Ausdrucks versteht man eine Variable x_i oder eine negierte Variable $\neg x_i$.

Ein Boolescher Ausdruck *F* heißt **erfüllbar**, wenn es eine Ersetzung der Variablen in *F* durch Werte 0 (*FALSCH*) bzw. 1 (*WAHR*) gibt, wobei selbstverständlich gleiche Variablen durch die gleichen Werte ersetzt werden (man sagt: die Variablen werden mit 0 bzw. 1 **belegt**), so daß sich bei Auswertung der so veränderten Formel *F* gemäß den Regeln über die Junktoren der Wert 1 (*WAHR*) ergibt.

Ein Boolescher Ausdruck *F* ist in **konjunktiver Normalform**, wenn *F* die Form

$F = F_1 \wedge \ldots \wedge F_m$

und jedes F_i die Form

$F_i = \left(y_{i_1} \vee \ldots y_{i_k}\right)$

hat, wobei y_{i_j} ein Literal oder eine Konstante ist, d.h. für eine Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 bzw. 1 steht.

Die Teilformeln F_i von *F* bezeichnet man als die **Klauseln** (von *F*). Natürlich ist eine Klausel selbst in konjunktiver Normalform.

Eine Klausel F_i von *F* ist durch eine Belegung der in *F* vorkommenden Variablen erfüllt, d.h. besitzt den Wahrheitswert *WAHR*, wenn mindestens ein Literal in F_i erfüllt ist d.h. den Wahrheitswert *WAHR* besitzt. *F* ist erfüllt, wenn alle in *F* vorkommenden Klauseln erfüllt sind.

---

[3] Die Syntax eines Booleschen Ausdrucks bzw. einer Formel der Aussagenlogik wird hier als bekannt vorausgesetzt.

Zu jedem Booleschen Ausdruck *F* gibt es einen Booleschen Ausdruck in konjunktiver Normalform, der genau dann erfüllbar ist, wenn *F* erfüllbar ist.

Das folgende Entscheidungsproblem liegt in **P**:

Erfüllende Wahrheitsbelegung

Instanz: $[F, f]$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$, $f : V \to \{WAHR, FALSCH\}$ ist eine Belegung der Variablen mit Wahrheitswerten.

$size([F, f]) = n$.

Lösung: Entscheidung „ja“, falls die durch *f* gegebene Belegung dem Ausdruck *F* den Wahrheitswert *WAHR* gibt,

Entscheidung „nein“, sonst.

Ein Lösungsalgorithmus setzt einfach die in der Eingabe-Instanz $[F, f]$ gelieferte Belegung *f* in die Formel *F* ein und ermittelt den Wahrheitswert der Formel gemäß den Auswertungsregeln für die beteiligten Junktoren.

Kodiert man einen Booleschen Ausdruck *F*, der *n* Variablen $x_1, ..., x_n$ und *m* Junktoren enthält, mit Hilfe des Alphabets $A = \{\wedge, \vee, \neg, (,), 0, 1, x\}$, so entsteht eine Zeichenkette der Länge $C \cdot (n \cdot \log(n) + m)$. Da jeder Junktor höchstens jeweils zwei neue Variablen verknüpft, ist $m \leq 2n$. Der Faktor $\log(n)$ entsteht nur durch die Unterscheidung der Variablen bzw. der Tatsache, daß zur Unterscheidung der Variablen nicht unterschiedliche Variablensymbole verwendet werden, um *F* mit Hilfe des endlichen Alphabets *A* zu kodieren. Man kann also annehmen, daß auch die Kodierung von $[F, f]$ durch eine Zeichenkette der Ordnung $O(n)$ erfolgt.

Das folgende Entscheidungsproblem liegt in **PSPACE**:

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT)

Instanz: F

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$.

$size(F) = n$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von *F* gibt, so daß sich bei Auswertung der Formel *F* der Wahrheitswert *WAHR* ergibt,
Entscheidung „nein“, sonst.

CSAT liegt in **PSPACE** (und damit in **EXP)**. Für den Beweis genügt es zu zeigen, daß nacheinander alle möglichen 2^n Belegungen der *n* Variablen in einer Eingabe-Instanz *F* mit einem Speicherplatzverbrauch von polynomiell vielen Speicherzellen erzeugt, in die Formel *F* eingesetzt und ausgewertet werden können.

Es ist nicht bekannt, ob CSAT in **P** liegt (vieles spricht dagegen).

Ein weiteres wichtiges Problem ist das folgende:

Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT)

Instanz: F
F ist ein Boolescher Ausdruck mit der Variablenmenge $V = \{x_1, ..., x_n\}$.
$size(F) = n$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von *F* gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert *WAHR* ergibt,
Entscheidung „nein“, sonst.

SAT liegt in **PSPACE** (und damit in **EXP)**.

Der folgende Spezialfall liegt jedoch in **P**:

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Instanz: F
$F = F_1 \wedge ... \wedge F_m$ ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält.
$size(F) = n$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von *F* gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert *WAHR* ergibt,
Entscheidung „nein“, sonst.

Ein Algorithmus zur Ermittlung einer erfüllenden Belegung kann etwa nach folgender Backtracking-Strategie vorgehen:

Man nehme eine bisher noch nicht betrachtete Klausel $F_i = (y_1 \vee y_2)$ von *F*. Falls eines der Literale während des bisherigen Ablaufs bereits den Wahrheitswert *WAHR* erhalten hat, dann wird F_i als erfüllt erklärt. Falls eines der Literale, etwa y_1, den Wert *FALSCH* hat, dann erhält das Literal y_2 den Wahrheitswert *WAHR* und $\neg y_2$ den Wahrheitswert *FALSCH*. *F* gilt dann als erfüllt. Dabei kann jedoch ein Konflikt auftreten, nämlich daß ein Literal durch die Zuweisung einen Wahrheitswert bekommen soll, jedoch den komplementären Wahrheitswert bereits vorher erhalten hat. In diesem Fall wird die vorherige Zuweisung rückgängig gemacht. Falls es dann wieder zu einem Konflikt mit diesem Literal kommt, ist die Formel nicht erfüllbar.

Der folgende Pseudocode-Algorithmus implementiert diese Strategie; aus Gründen der Lesbarkeit wird auf die exakte Deklaration der lokalen Variablen und der verwendeten Datentypen und auf deren Implementierung verzichtet.

Algorithmus zur Lösung des Erfüllbarkeitsproblems für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Eingabe: $F = F_1 \wedge ... \wedge F_m$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält, d.h. jedes F_i hat die Form $F_i = (y_{i_1} \vee y_{i_2})$

$size(F) = n$.

Der Datentyp
`TYPE KNF_typ = ...;`
beschreibe den Typ einer Formel in konjunktiver Normalform, der Datentyp
`TYPE Literal_typ = ...;`
den Typ eines Literals bzw. einer Variablen in einem Booleschen Ausdruck.

```
VAR F            : KNF_typ;
    Entscheidung : Entscheidungs_typ;
```

`F :=` $F_1 \wedge ... \wedge F_m$

Verfahren: Aufruf der Prozedur
`Erfuellbarkeit_2CSAT (F, Entscheidung);`

Im Ablauf der Prozedur werden Variablen Wahrheitswerte *WAHR (TRUE)* bzw. *FALSCH (FALSE)* zugewiesen. Zu Beginn des Ablaufs hat jede Variable noch einen undefinierten Wert; in diesem Fall wird die Variable als „noch nicht

zugewiesen“ bezeichnet. Jede Klausel F_i von F gilt zu Beginn des Prozedurablaufs als „unerfüllt“ (da ihren Literalen ja noch kein Wahrheitswert zugewiesen wurde). Sobald einem Literal in F_i der Wahrheitswert *TRUE* zugewiesen wurde, gilt die Klausel als „erfüllt“.

Ausgabe: `Entscheidung = ja`, falls F erfüllbar ist,
`Entscheidung = nein` sonst.

```
PROCEDURE Erfuellbarkeit_2CSAT (F                : KNF_typ;
                                VAR Entscheidung:Entscheidungs_typ);
VAR C          : SET OF KNF_typ;
    V          : SET OF Literal_typ;
    x          : Literal_typ;
    firstguess : BOOLEAN;

BEGIN { Erfuellbarkeit_2CSAT }
  { F = F_1 ∧ ... ∧ F_m }
  C := {F_1, ..., F_m};
  erkläre alle Klauseln in C als unerfüllt;
  V := Menge der in F vorkommenden Variablen;
  erkläre alle Variablen in V als nicht zugewiesen;

  WHILE (V enthält eine Variable x) DO
    BEGIN
      x          := TRUE;
      firstguess := TRUE;
      WHILE (C enthält eine unerfüllte Klausel F_i = (y_i1 ∨ y_i2),
             wobei mindestens einem Literal ein Wahrheitswert
             zugewiesen ist) DO
         BEGIN
           IF (y_i1 = TRUE) OR (y_i2 = TRUE)
           THEN erkläre F_i als erfüllt
           ELSE IF (y_i1 = FALSE) AND (y_i2 = FALSE)
                THEN BEGIN
                       IF NOT firstguess
                       THEN BEGIN
                              Entscheidung := nein;
                              Exit
                            END
                       ELSE BEGIN
                              erkläre alle Klauseln in C
                                als unerfüllt;
```

```
                              erkläre alle Variablen in V
                                als nicht zugewiesen;
                              x          := FALSE;
                              firstguess := FALSE;
                            END;
                      END
                ELSE BEGIN
                       IF  y_{i_1}  = FALSE
                       THEN  y_{i_2}  := TRUE
                       ELSE  y_{i_1}  := TRUE;
                       erkläre  F_i  als erfüllt;
                      END;
          END { WHILE C enthält eine unerfüllte Klausel };
       entferne aus C die erfüllten Klauseln;
       entferne aus V die Variablen, denen ein Wahrheitswert
                                     zugewiesen wurde;
    END { WHILE V enthält eine Variable x};
  Entscheidung := ja;

END   { Erfuellbarkeit_2CSAT };
```

Ist F eine Instanz für 2-CSAT mit n Variablen und m Klauseln, $size(F) = n$, dann liefert das beschriebene Verfahren mit der Prozedur `Erfuellbarkeit_2CSAT` eine korrekte ja/nein-Entscheidung. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n \cdot m)$. Wegen $m = O(n)$ ist dieser Aufwand quadratisch, d.h. polynomiell in der Größe der Eingabe.

Für das zu 2-CSAT analoge Problem 3-CSAT der Erfüllbarkeit, bei dem jede Klausel genau 3 Literale enthält, ist kein polynomielles Entscheidungsverfahren bekannt. Es wird vermutet, daß es auch kein derartiges Verfahren gibt. Natürlich liegt 3-CSAT in **PSPACE** und damit in **EXP**.

Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) bzw. das Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT) mit einer Eingabeinstanz F und das Problem der erfüllenden Wahrheitsbelegung mit einer Eingabeinstanz $[F, f]$ unterscheiden sich grundsätzlich dadurch, daß bei letzterem in der Eingabeinstanz $[F, f]$ eine wesentliche Zusatzinformation, nämlich eine potentiell erfüllende Belegung f der Variablen vorgegeben ist, die nur noch daraufhin überprüft werden muß, ob sie wirklich die in der Eingabeinstanz enthaltene Formel F erfüllt. Zur Entscheidung, ob eine Eingabeinstanz F von CSAT erfüllbar ist, muß also entweder eine erfüllende Belegung f kon-

struiert bzw. aufgrund geeigneter Argumente die Erfüllbarkeit gezeigt werden, oder es muß der Nachweis erbracht werden, daß keine Belegung der Variablen von *F* die Formel erfüllt. Wenn dieser Nachweis nur dadurch gelingt, daß *alle* 2^n möglichen Belegungen überprüft werden, ist mit einem polynomiellen Entscheidungalgorithmus nicht zu rechnen. Intuitiv ist diese Entscheidungsaufgabe also schwieriger zu bewältigen, weil weniger Anfangsinformationen vorliegen, als lediglich die Verifikation einer potentiellen Lösung. Diese Überlegung führt auf die Definition eines weiteren Typs von Algorithmen.

Es sei Π ein Entscheidungsproblem:

Instanz: $x \in \Sigma_\Pi^*$
und Spezifikation einer Eigenschaft, die einer Auswahl von Elementen aus Σ^* zukommt, d.h. die Spezifikation einer Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit
$L_\Pi = \{u \in \Sigma^* \mid u \text{ hat die beschriebene Eigenschaft}\}$

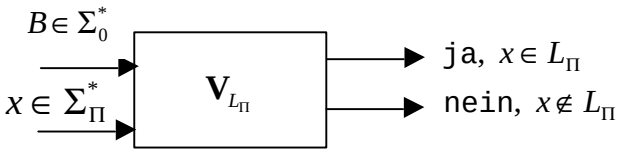
Lösung: Entscheidung „ja“, falls $x \in L_\Pi$ ist,
Entscheidung „nein“, falls $x \notin L_\Pi$ ist.

Ein **Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π ist ein Algorithmus $\mathbf{V}_{L_\Pi}$, der eine Eingabe $x \in \Sigma_\Pi^*$ und eine **Zusatzinformation** (einen **Beweis**) $B \in \Sigma_0^*$ lesen kann und die Frage „$x \in L_\Pi$?“ mit Hilfe des Beweises *B* entscheidet. Die ja/nein-Entscheidung, die ein Verifizierer $\mathbf{V}_{L_\Pi}$ bei Eingabe von *x* und *B* trifft, wird mit $\mathbf{V}_{L_\Pi}(x,B)$ bezeichnet.

Ein **Verifizierer $\mathbf{V}_{L_\Pi}$ für das Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$

Ausgabe: ja, falls $x \in L_\Pi$ gilt
nein, falls $x \notin L_\Pi$ gilt.



Mit $\mathbf{V}_{L_\Pi}(x,B)$, $\mathbf{V}_{L_\Pi}(x,B) \in \{\texttt{ja}, \texttt{nein}\}$, wird die Entscheidung von $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ und $B \in \Sigma_0^*$ bezeichnet. Zu beachten ist, daß $\mathbf{V}_{L_\Pi}$ bei jeder Eingabe $x \in \Sigma_\Pi^*$ hält.

Die Aufgabe eines Verifizierers $\mathbf{V}_{L_\Pi}$ für ein Entscheidungsproblem Π besteht also darin, bei Eingabe von $x \in \Sigma_\Pi^*$ zu entscheiden, ob $x \in L_\Pi$ gilt oder nicht. Dazu bedient er sich jeweils eines geeigneten Beweises $B \in \Sigma_0^*$. Ein Beweis ist dabei eine Zeichenkette über einem Alphabet, das der Problemstellung angemessen ist. Häufig wird ein Beweis als Wort über dem Alphabet $\Sigma_0 = \{0,1\}$ formuliert. Wie ein derartiger Beweis bei Eingabe von $x \in \Sigma_\Pi^*$ zu konstruieren ist, liegt außerhalb des Algorithmus $\mathbf{V}_{L_\Pi}$. Man sagt, ein Beweis *B* wird **auf nichtdeterministische Weise** vorgegeben.

Mit Hilfe eines Verifizierers $\mathbf{V}_{L_\Pi}$ läßt sich ein **nichtdeterministischer Algorithmus zur Akzeptanz (Erkennung, Entscheidung)** der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ definieren:

Ein **nichtdeterministischer Algorithmus zur Akzeptanz (Erkennung, Entscheidung)** der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit Hilfe eines Verifizierers $\mathbf{V}_{L_\Pi}$ hat die Schnittstelle:

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt mit
$\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$;
Entscheidung „$x \notin L_\Pi$“ falls für alle Beweise $B \in \Sigma_0^*$ gilt: $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Analog zur Arbeitsweise eines zeitbeschränkten deterministischen Algorithmus kann man auch zeitbeschränkte nichtdeterministische Algorithmen definieren:

Es sei $f : \mathbf{N} \to \mathbf{N}$ eine Funktion. Falls der nichtdeterministische Algorithmus mit Hilfe des Verifizierers $\mathbf{V}_{L_\Pi}$ die ja/nein-Entscheidung bei Eingabe von $x \in \Sigma_\Pi^*$ mit $|x| = n$ nach einer Anzahl von Schritten trifft, die in $O(f(n))$ liegt, so ist der Algorithmus (der Verifizierer $\mathbf{V}_{L_\Pi}$) $f(n)$-**zeitbeschränkt**.

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in NTIME(f(n))$, wenn es einen nichtdeterministischen $f(n)$-zeitbeschränkten Algorithmus zur Akzeptanz von L_Π gibt.

Die Aussage
„das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $NTIME(f(n))$“
steht synonym für $L_\Pi \in NTIME(f(n))$. In diesem Sinn bezeichnet $NTIME(f(n))$ die **Klasse der Entscheidungsprobleme, die auf nichtdeterministische Weise in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Zu beachten ist, daß die Laufzeit des nichtdeterministischen Algorithmus bzw. die Laufzeit des Verifizierers $\mathbf{V}_{L_\Pi}$ in Abhängigkeit von der Größe von $x \in \Sigma_\Pi^*$ gemessen wird. Natürlich ist im Zeitaufwand, den ein Verifizierer für seine Entscheidung benötigt, auch die Zeit enthalten, um die Zeichen des Beweises *B* zu lesen. Ist dieser $f(n)$-zeitbeschränkt, so kann er höchstens $C \cdot f(n)$ viele Zeichen des Beweises lesen (hierbei ist *C* eine Konstante). Wenn es also überhaupt einen Beweis B_x mit $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$ gibt, dann gibt es auch einen Beweis mit Länge $\leq C \cdot f(n)$, so daß man für den Beweis gleich eine durch $C \cdot f(n)$ beschränkte Länge annehmen kann.

Eine der wichtigsten Klassen, die Klasse **NP** der Entscheidungsprobleme, die nichtdeterministisch mit polynomieller Zeitkomplexität gelöst werden können, ergibt sich, wenn *f* ein Polynom ist:

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right)$$

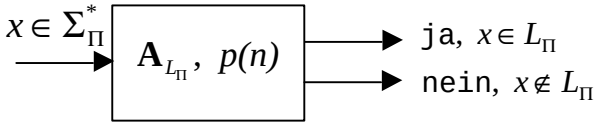
Zur Verdeutlichung des Unterschieds zwischen **P** und **NP** werden die entsprechenden Definitionen der beiden Klassen noch einmal gegenübergestellt:

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **P**, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{ja}$ gilt,
Entscheidung „$x \notin L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{nein}$ gilt.

Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.



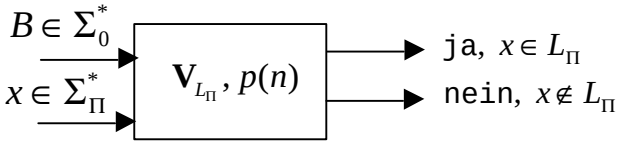
Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **NP**, wenn es einen Verifizierer $\mathbf{V}_{L_\Pi}$ und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt mit $|B_x| \le C \cdot p(n)$ und $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$;

Entscheidung „$x \notin L_\Pi$“ falls für alle Beweise $B \in \Sigma_0^*$ mit $|B| \le C \cdot p(n)$ gilt: $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Hierbei wird die Entscheidung $\mathbf{V}_{L_\Pi}(x, B)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.



Da ein polynomiell zeitbeschränkter Algorithmus $\mathbf{A}_{L_\Pi}$, wie er in der Definition von **P** vorkommt, ein spezieller polynomiell zeitbeschränkter Verifizierer ist, der für seine Entscheidung ohne die Zuhilfenahme eines Beweises auskommt, gilt

$\mathbf{P} \subseteq \mathbf{NP}$.

Die Frage $\mathbf{P} = \mathbf{NP}$ bzw. $\mathbf{P} \neq \mathbf{NP}$ ist bisher ungelöst (**P-NP-Problem**). Vieles spricht dafür, daß $\mathbf{P} \neq \mathbf{NP}$ gilt.

Im folgenden werden einige wenige **Beispiele für Probleme Π aus NP** aufgeführt, von denen nicht bekannt ist , ob sie in **P** liegen. Dabei wird für Instanzen $x \in L_\Pi$ jeweils der Beweis $B_x \in \Sigma_0^*$ angegeben, der den Verifizierer zur `ja`-Entscheidung veranlaßt. Die Angabe von B_x erfolgt informell, sie muß entsprechend (beispielsweise in eine binäre Zeichenkette) übersetzt werden.

- Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) bzw. Boolescher Ausdrücke in allgemeiner Form (SAT):
  Instanz: F ist ein Boolescher Ausdruck in konjunktiver Normalform bzw. in allgemeiner Form mit der Variablenmenge $V = \{x_1, ..., x_n\}$; $size(F) = n$.

Beweis: B_F ist eine Kodierung der Belegung der in *F* vorkommenden Variablen, z.B. als 0-1-Folge der Länge *n*.

Arbeitsweise des Verifizierers: Einsetzen der Belegung B_F in die Variablen von *F* und Auswertung der Formel.

- Problem des Handlungsreisenden:

  Instanz: $[G,K]$, $G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{\geq 0}$; $size([G,K]) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$.

  Beweis: $B_{[G,K]}$ ist eine Permutation der Zahlen 1, ..., *n* in Binärkodierung (mit Länge in $O(n \cdot \log(n))$).

  Arbeitsweise des Verifizierers: Überprüfung, ob die Permutation $B_{[G,K]}$ eine Rundreise in G beschreibt, deren Gewicht $\leq K$ ist. Mit Binärsuche wie in Kapitel 3.1 beschrieben läßt sich sogar in polynomieller Zeit überprüfen, ob es eine Rundreise mit minimalen Gewicht $\leq K$ gibt.

- Partitionenproblem:

  Instanz: $I=\{a_1,...,a_n\}$

  I ist eine Menge von *n* natürlichen Zahlen; $size(I) = n \cdot \log_2(B)$ mit $B = \sum_{i=1}^{n} a_i$.

  Beweis: B_I ist eine Teilmenge von $\{1,...,n\}$ (mit Länge in $O(n \cdot \log(n))$).

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{j \in B_I} a_j = \sum_{j \notin B_I} a_j$ gilt.

- 0/1-Rucksackproblem:

  Instanz: $I=(A,M)$

  $A=\{a_1,...,a_n\}$ ist eine Menge von *n* natürlichen Zahlen; $M \in \mathbf{N}$; $size(I) = n \cdot \log_2(a_n)$

  Beweis: B_I ist eine Folge $x_1,...,x_n$ von *n* Zahlen aus $\{0,1\}$ (mit Länge *n*).

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{i=1}^{n} x_i \cdot a_i = M$ gilt.

- Problem der minimalen $\{0,1\}$-Linearen Programmierung

  Instanz: $[A,\vec{b},\vec{c},K]$, $A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit *m* Zeilen und *n* Spalten, $\vec{b} \in \mathbf{Z}^m$ ist ein ganzzahliger Vektor, $\vec{c} \in \mathbf{N}^n$ ist ein nichtnegativer ganzzahliger Vektor, $K \in \mathbf{N}$, $size([A,\vec{b},\vec{c}]) = k = m \cdot n \cdot B$ mit

$$B = \max\left\{\lceil \log(e) \rceil \mid e \in A \vee e \in \vec{b} \vee e \in \vec{c}\right\}.$$

Gesucht: Vektor $\vec{x} \in \{0,1\}^n$ mit $A \cdot \vec{x} \geq \vec{b}$ und Wert $\sum_{i=1}^{n} c_i \cdot x_i \leq K$.

Beweis: $B_{[A,\vec{b},\vec{c}]}$ ist eine 0-1-Folge der Länge *n* (für den Vektor $\vec{x}$).

Arbeitsweise des Verifizierers: Überprüfung, ob für $\vec{x} = B_{[A,\vec{b},\vec{c}]}$ die Bedingungen $A \cdot \vec{x} \geq \vec{b}$ und $\sum_{i=1}^{n} c_i \cdot x_i \leq K$ gelten.

- Problem des längsten Wegs in einem gewichteten Graphen:

  Instanz: $[G, v_i, v_j, K]$, $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; $v_i \in V$; $v_j \in V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{\geq 0}$; $size([G, K]) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$.

  Gesucht: ein einfacher Weg (d.h. ohne Knotenwiederholungen) von v_i nach v_j mit Gewicht $\geq K$.

  Beweis: $B_{[G,v_i,v_j,K]}$ ist eine Folge von Knoten, die mit v_i beginnt und mit v_j endet (mit Länge in $O(n \cdot \log(n))$.

  Arbeitsweise des Verifizierers: Überprüfung, ob $B_{[G,v_i,v_j,K]}$ einen einfachen Weg beschreibt, dessen Gewicht $\geq K$ ist.

Man kennt heute mehrere tausend Probleme aus **NP**. In der angegebenen Literatur werden geordnet nach Anwendungsgebieten umfangreiche Listen von **NP**-Problemen aufgeführt.

Die Arbeitsweise eines nichtdeterministischen polynomiellen Entscheidungsalgorithmus kann man durch einen „normalen“ deterministischen Algorithmus simulieren. Es ist jedoch bisher keine Simulation bekannt, die dabei in polynomieller Zeit arbeitet. Genauer gilt:

Für jedes Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ aus **NP** mit Verifizierer $\mathbf{V}_{L_\Pi}$ und polynomieller Zeitkomplexität $p(n)$ gibt es einen deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$, der für jedes $x \in \Sigma_\Pi^*$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht und die Zeitkomplexität $O\left(p(n) \cdot 2^{O(p(n))}\right)$ besitzt.

Die Aussage „**P** = **NP**“ würde bedeuten, daß es für jedes Problem Π aus **NP** mit polynomiell zeitbeschränktem Verifizierer $\mathbf{V}_{L_\Pi}$ sogar einen *polynomiell zeitbeschränkten* deterministi-

schen Algorithmus $\mathbf{A}_{L_\Pi}$ gibt, der dasselbe Ergebnis wie $\mathbf{V}_{L_\Pi}$ liefert. Das bzw. die Unmöglichkeit einer derartigen Simulation sind jedoch nicht bekannt.

Der folgende Pseudocode für $\mathbf{A}_{L_\Pi}$ beschreibt die wesentlichen Aspekte der Simulation:

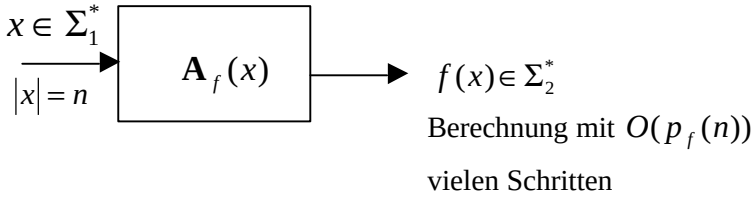
Bemerkung: Ein Beweis *B*, den der Verifizierer liest, ist ein Wort über einem Alphabet Σ_0 .

```
FUNCTION A_{L_Π}(x);

VAR n       : INTEGER;
    lng     : INTEGER;
    B       : Σ0*;
    antwort : BOOLEAN;
    weiter  : BOOLEAN;

BEGIN
  n       := size(x);      { Größe der Eingabe             }
  lng     := C * p(n);     { maximale Länge eines Beweises }
  antwort := FALSE;
  weiter  := TRUE;

  WHILE weiter DO
    IF (es sind bereits alle Wörter aus Σ0* mit Länge <= lng
        erzeugt worden)
    THEN weiter := FALSE
    ELSE
      BEGIN
        B:= nächstes Wort aus Σ0* mit Länge <= lng;
        IF V_Π(x,B) = ja
        THEN BEGIN
               antwort := TRUE;
               weiter  := FALSE;
             END;
      END;

  CASE antwort OF
  TRUE  : A_Π(x) := ja;
  FALSE : A_Π(x) := nein;
  END;

END;
```

3.4 NP-Vollständigkeit

Die bisher beschriebenen Probleme entstammen verschiedenen Anwendungsgebieten. Dementsprechend unterscheiden sich die Alphabete, mit denen man die jeweiligen Instanzen bildet. Boolesche Ausdrücke werden mit einem anderen Alphabet kodiert als gewichtete Graphen oder Instanzen für das Partitionenproblem. Im folgenden wird zunächst eine Verbindung zwischen den unterschiedlichen Problemen und ihren zugrundeliegenden Alphabeten hergestellt.

Eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$, die Wörter über dem endlichen Alphabet Σ_1 auf Wörter über dem endlichen Alphabet Σ_2 abbildet, heißt **durch einen deterministischen Algorithmus in polynomieller Zeit berechenbar**, wenn gilt: Es gibt einen deterministischen Algorithmus $\mathbf{A}_f$ mit Eingabemenge Σ_1^* und Ausgabemenge Σ_2^* und ein Polynom p_f mit folgenden Eigenschaften:
bei Eingabe von $x \in \Sigma_1^*$ mit der Größe $|x| = n$ erzeugt der Algorithmus die Ausgabe $f(x) \in \Sigma_2^*$ und benötigt dazu höchstens $O(p_f(n))$viele Schritte.



Es seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Mengen aus Zeichenketten (Wörtern) über jeweils zwei endlichen Alphabeten. L_1 heißt **polynomiell (many-one) reduzierbar** auf L_2, geschrieben
$L_1 \leq_m L_2$
wenn gilt: Es gibt eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$, die durch einen deterministischen Algorithmus in polynomieller Zeit berechenbar ist und für die gilt:
$w \in L_1 \Leftrightarrow f(w) \in L_2$.

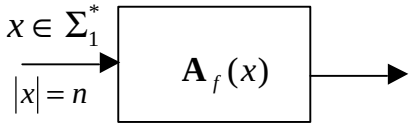
Diese Eigenschaft kann auch so formuliert werden:
$w \in L_1 \Rightarrow f(w) \in L_2$ und $w \notin L_1 \Rightarrow f(w) \notin L_2$.

Bemerkung: Es gibt noch andere Formen der Reduzierbarkeit zwischen Mengen, z.B. die allgemeinere Form der Turing-Reduzierbarkeit mit Hilfe von Orakel-Turingmaschinen.

Die **Bedeutung der Reduzierbarkeit** $\leq_m$ zeigt folgende Überlegung.

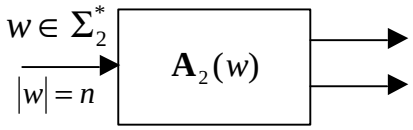
Für die Mengen $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ gelte $L_1 \leq_m L_2$ mittels der Funktion $f : \Sigma_1^* \to \Sigma_2^*$ bzw. des Algorithmus $\mathbf{A}_f$. Seine Zeitkomplexität sei das Polynom p_f. Für die Menge L_2 gebe es einen polynomiell zeitbeschränkten deterministischen Algorithmus $\mathbf{A}_2$, der L_2 erkennt; seine Zeitkomplexität sei das Polynom p_2 :

Berechnung von *f*:



Berechnung mit $O(p_f(n))$ vielen Schritten

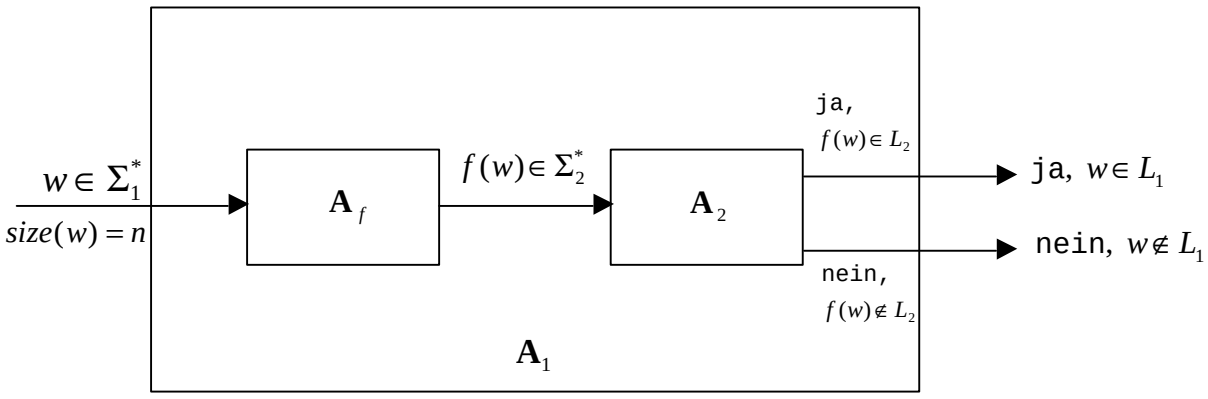
Erkennung von L_2:



Entscheidung nach $O(p_2(n))$ vielen Schritten

Mit Hilfe von $\mathbf{A}_f$ und $\mathbf{A}_2$ läßt sich durch Hintereinanderschaltung beider Algorithmen ein polynomiell zeitbeschränkter deterministischer Algorithmus $\mathbf{A}_1$ konstruieren, der L_1 erkennt:

Erkennung von L_1:



Erkennung von L_1 mit Zeitaufwand der Größenordnung $O\left(p_f(n) + p_2\left(p_f(n)\right)\right)$, d.h.

Ein $w_2 \in \Sigma_2^*$ kann dazu dienen, für mehrere (*many*) $w \in \Sigma_1^*$ die Frage „$w \in L_1$?“ zu entscheiden (nämlich für alle diejenigen $w \in \Sigma_1^*$, für die $f(w) = w_2$ gilt). Allerdings darf man für jede Eingabe $f(w)$ den Algorithmus $\mathbf{A}_2$ nur einmal (*one*) verwenden, nämlich bei der Entscheidung von $f(w)$.

Gilt für ein Problem Π_0 zur Entscheidung der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ und für ein Problem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ die Relation $L_\Pi \leq_m L_{\Pi_0}$, so heißt das Problem Π auf das Problem Π_0 **polynomiell (many-one) reduzierbar**, geschrieben $\Pi \leq_m \Pi_0$.

Beispiel:

$\{0,1\}$-Lineare Programmierung:

Instanz: $[A, \vec{b}, \vec{z}]$

$A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit *m* Zeilen und *n* Spalten, $\vec{b} \in \mathbf{Z}^m$, $\vec{z}$ ist ein Vektor von *n* Variablen, die die Werte 0 oder 1 annehmen können.

Lösung: Entscheidung „ja“, falls gilt:
Es gibt eine Zuweisung der Werte 0 oder 1 an die Variablen in $\vec{z}$, so daß das lineare Ungleichungssystem $A \cdot \vec{z} \leq / \geq \vec{b}$ erfüllt ist. Die Bezeichnung $\leq / \geq$ steht hier für entweder $\leq$ oder $\geq$ in einer Ungleichung.

Es gilt CSAT $\leq_m$ $\{0,1\}$-Lineare Programmierung.

Eine der zentralen Definitionen in der Angewandten Komplexitätstheorie ist die **NP**-Vollständigkeit:

Ein Entscheidungsproblem Π_0 zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ heißt **NP-vollständig**, wenn gilt:
$L_{\Pi_0} \in \mathbf{NP}$, und für jedes Probleme Π zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit $L_\Pi \in \mathbf{NP}$ gilt $L_\Pi \leq_m L_{\Pi_0}$.

Mit obiger Definition der Reduzierbarkeit zwischen Problemen kann man auch sagen:

Das Entscheidungsproblem Π_0 ist **NP**-vollständig, wenn Π_0 in **NP** liegt und für jedes Entscheidungsprobleme Π in **NP** die Relation $\Pi \leq_m \Pi_0$ gilt.

Das erste Beispiel eines **NP**-vollständigen Problems wurde 1971 von Stephen Cook gefunden. Es gilt:

Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT) ist **NP**-vollständig.

Die Beweisidee (hier nicht direkt dem Beweis von Cook folgend) läßt sich folgendermaßen skizzieren:

1. SAT liegt in **NP** (siehe Kapitel 3.3).
2. Für jedes Problem Π in **NP** ist die Relation $\Pi \leq_m \mathrm{SAT}$ zu zeigen. Dazu geht man von dem nichtdeterministischen polynomiell zeitbeschränkten Algorithmus zur Akzeptanz der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit Hilfe eines Verifizierers $\mathbf{V}_{L_\Pi}$ aus und beschreibt durch einen Booleschen Ausdruck $F(\Pi)$ die Arbeitsweise von $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ mit $size(x) = n$ und polynomiell in *n* beschränktem Beweis $B \in \Sigma_0^*$. Die Länge von $F(\Pi)$ läßt sich polynomiell in *n* beschränken, so daß die Konstruktion von $F(\Pi)$ nur polynomiellen Zeitaufwand erfordert. Dieser Boolesche Ausdruck $F(\Pi)$ beschreibt im Prinzip, daß $\mathbf{V}_{L_\Pi}$ in einem Anfangszustand startet, welche Inhalte die Speicherzellen haben, die während des Ablaufs des Algorithmus von $\mathbf{V}_{L_\Pi}$ verwendet und verändert werden, welche Rechenschritte (Zustandsänderungen) im Algorithmus erfolgen und daß das Verfahren nach polynomieller Zeit stoppt. Zusätzlich wird $F(\Pi)$ so konstruiert, daß gilt: $x \in L_\Pi \Leftrightarrow F[\Pi]$ ist erfüllbar.

NP-vollständige Probleme kann man innerhalb der Klasse **NP** als die am schwersten zu lösenden Probleme betrachten, denn sie entscheiden die **P**-**NP**-Frage:

Gibt es mindestens ein **NP**-vollständiges Problem, das in **P** liegt , so ist **P** = **NP**.

Aus **NP**-vollständigen Problemen lassen sich aufgrund der Transitivität der $\leq_m$-Relation weitere **NP**-vollständige Probleme ableiten:

Ist das Entscheidungsproblem Π_0 **NP**-vollständig und gilt $\Pi_0 \leq_m \Pi_1$ für ein Entscheidungsproblem Π_1 so gilt: Ist Π_1 in **NP**, so ist Π_1 ebenfalls **NP**-vollständig.

Bei $\mathbf{P} \neq \mathbf{NP}$ gilt:

Ist ein Entscheidungsproblem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ **NP**-vollständig, so ist das Problem Π schwer lösbar (intractable), d.h. es gibt keinen polynomiell zeitbeschränkten deterministischen Lösungsalgorithmus zur Entscheidung von L_Π. Insbesondere ist das zugehörige Optimierungsproblem, falls es ein solches gibt, erst recht schwer (d.h. nur mit mindestens exponentiellem Aufwand) lösbar.

Man weiß von vielen praktisch relevanten Entscheidungsproblemen Π, daß sie **NP**-vollständig sind.

NP-vollständige Probleme mit praktischer Relevanz sind heute aus vielen Gebieten bekannt, z.B. aus der Graphentheorie, dem Netzwerk-Design, der Theorie von Mengen und Partitionen, der Datenspeicherung, dem Scheduling, der Maschinenbelegung, der Personaleinsatzplanung, der mathematischen Programmierung und Optimierung, der Analysis und Zahlentheorie, der Kryptologie, der Logik, der Automatentheorie und der Theorie Formaler Sprachen, der Programm- und Codeoptimierung usw. (siehe Literatur). Die folgende Zusammenstellung listet einige wenige Beispiele auf, die z.T. bereits behandelt wurden.

Beispiele für NP-vollständige Entscheidungsprobleme

- **Erfüllbarkeitsproblem der Aussagenlogik (SAT):**

  Instanz: *F*,
  F ist ein Boolescher Ausdruck (Formel der Aussagenlogik)

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar, d.h. es gibt eine Belegung der Variablen in *F* mit Werten 1 (*TRUE*, *WAHR*) bzw. 0 (*FALSE*, *FALSCH*), wobei gleiche Variablen mit gleichen Werten belegt werden, so daß sich bei Auswertung der Formel *F* der Wert 1 (*TRUE*, *WAHR*) ergibt.

  Kodiert man Formeln der Aussagenlogik über dem Alphabet $A = \{\wedge, \vee, \neg, (,), x, 0, 1\}$, so ist eine Formel der Aussagenlogik ein Wort über dem Alphabet *A*. Nicht jedes Wort über dem Alphabet *A* ist eine Formel der Aussagenlogik (weil es eventuell syntaktisch nicht korrekt ist) und nicht jede Formel der Aussagenlogik als Wort über dem Alphabet *A* ist erfüllbar.

  $L_{\mathrm{SAT}} = \{F \mid F \text{ ist eine } \mathbf{erfüllbare} \text{ Formel der Aussagenlogik}\}$.

- **Erfüllbarkeitsproblem der Aussagenlogik mit Formeln in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel (3-CSAT):**

  Instanz: F,
  F ist eine Formel der Aussagenlogik in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel, d.h. sind $x_1, ..., x_n$ die verschiedenen in F vorkommenden Booleschen Variablen, so hat F die Form
  $F = F_1 \wedge F_2 \wedge ... F_m$
  Hierbei hat jedes F_i die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ oder $F_i = (y_{i_1} \vee y_{i_2})$ oder $F_i = (y_{i_1})$, und y_{i_j} steht für eine Boolesche Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Boolesche Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 (d.h. $y_{i_j} = 0$) bzw. 1 (d.h. $y_{i_j} = 1$).

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar.

  Kodiert man die Formeln wie bei SAT, so gilt $L_{\text{3-CSAT}} \subseteq L_{\text{SAT}}$.

- **Kliquenproblem (KLIQUE):**

  Instanz: $[G, k]$,
  $G = (V, E)$ ist ein ungerichteter Graph und k eine natürliche Zahl.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt eine „Klique“ der Größe k. Dieses ist eine Teilmenge $V' \subseteq V$ der Knotenmenge mit $|V'| = k$, und für alle $u \in V'$ und alle $v \in V'$ mit $u \neq v$ gilt $(u, v) \in E$.

- **0/1-Rucksack-Entscheidungsproblem (RUCKSACK):**

  Instanz: $[a_1, ..., a_n, b]$,
  $a_1, ..., a_n$ und b sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, ..., n\}$ mit $\sum_{i \in I} a_i = b$.

- **Partitionenproblem (PARTITION):**

  Instanz: $[a_1, \ldots, a_n]$,
  $a_1, \ldots, a_n$ sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

- **Packungsproblem (BINPACKING):**

  Instanz: $[a_1, \ldots, a_n, b, k]$,
  $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i \leq b$ für $i = 1, \ldots, n$, $b \in \mathbf{N}$ („Behältergröße“), $k \in \mathbf{N}$.

  Lösung: Entscheidung „ja“, falls gilt:
  Die Objekte können so auf k Behälter der Füllhöhe b verteilt werden, so daß kein Behälter überläuft, d.h. es gibt eine Abbildung $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$, so daß für alle $j \in \{1, \ldots, k\}$ gilt: $\sum_{f(i)=j} a_i \leq b$

- **Problem des Hamiltonschen Kreises in einem gerichteten (bzw. ungerichteten) Graphen (GERICHTETER bzw. UNGERICHTETER HAMILTONKREIS):**

  Instanz: G,
  $G = (V, E)$ ist ein gerichteter bzw. ungerichteter Graph mit $V = \{v_1, \ldots, v_n\}$.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt einen Hamiltonschen Kreis. Dieses ist eine Anordnung $(v_{p(1)}, v_{p(2)}, \ldots, v_{p(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so daß für $i = 1, \ldots, n-1$ gilt: $(v_{p(i)}, v_{p(i+1)}) \in E$ und $(v_{p(n)}, v_{p(1)}) \in E$.

- **Problem des Handlungsreisenden (HANDLUNGSREISENDER):**

  Instanz: $[M, k]$,
  $M = (M_{i,j})$ ist eine $(n \times n)$-Matrix von „Entfernungen“ zwischen n „Städten“

und eine Zahl k.

Lösung: Entscheidung „ja“, falls gilt:
Es gibt eine Permutation π (eine Tour, „Rundreise“), so daß $\sum_{i=1}^{n-1} M_{p(i),p(i+1)} + M_{p(n),p(1)} \leq k$ gilt?

Zum Nachweis der **NP**-Vollständigkeit für eines dieser Probleme Π ist neben der Zugehörigkeit von Π zu **NP** jeweils die Relation $\Pi_0 \leq_m \Pi$ zu zeigen, wobei hier Π_0 ein Problem ist, für das bereist bekannt ist, daß es **NP**-vollständig ist. Häufig beweist man
SAT $\leq_m$ 3-CSAT $\leq_m$ RUCKSACK $\leq_m$ PARTITION $\leq_m$ BIN PACKING und
3-CSAT $\leq_m$ KLIQUE und
3-CSAT $\leq_m$ GERICHTETER HAMILTONKREIS $\leq_m$ UNGERICHTETER HAMILTONKREIS $\leq_m$ HANDUNGSREISENDER.

4 Approximation von Optimierungsaufgaben

Gegeben sei das Optimierungsproblem Π:

Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ eine Menge zulässiger Lösungen zuordnet
3. Spezifikation einer Zielfunktion m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ den Wert $m_\Pi(x, y)$ einer zulässigen Lösung zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$) bzw. $m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

Im folgenden (und in den vorhergehenden Beispielen) werden Optimierungsproblemen untersucht, die auf der Grenze zwischen praktischer Lösbarkeit (tractability) und praktischer Unlösbarkeit (intractability) stehen. In Analogie zu Entscheidungsproblemen in **NP** bilden diese die Klasse **NPO**:

Das Optimierungsproblem Π gehört zur **Klasse NPO**, wenn gilt:

1. die Menge der Instanzen $x \in \Sigma_\Pi^*$ ist in polynomieller Zeit entscheidbar
2. es gibt ein Polynom q mit der Eigenschaft: für jedes $x \in \Sigma_\Pi^*$ und jede zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ gilt $|y| \le q(|x|)$, und für jedes y mit $|y| \le q(|x|)$ ist in polynomieller Zeit entscheidbar, ob $y \in \mathrm{SOL}_\Pi(x)$ ist
3. die Zielfunktion m_Π ist in polynomieller Zeit berechenbar.

Alle bisher behandelten Beispiele für Optimierungsprobleme liegen in der Klasse **NPO**. Dazu gehören insbesondere auch solche, deren zugehöriges Entscheidungsproblem **NP**-vollständig ist.

Es gilt der Satz:

Für jedes Optimierungsproblem Π in **NPO** ist das zugehörige Entscheidungsproblem in **NP**.

Analog zur Definition der Klasse **P** innerhalb **NP** läßt sich innerhalb **NPO** eine Klasse **PO** definieren:

Ein Optimierungsproblem Π aus **NPO** gehört zur **Klasse PO**, wenn es einen polynomiell zeitbeschränkten Algorithmus gibt, der für jede Instanz $x \in \Sigma_\Pi^*$ eine optimale Lösung $y^* \in \mathrm{SOL}_\Pi(x)$ zusammen mit dem optimalen Wert $m_\Pi^*(x)$ der Zielfunktion ermittelt.

Offensichtlich ist $\mathbf{PO} \subseteq \mathbf{NPO}$.

Es gilt:

Ist $\mathbf{P} \neq \mathbf{NP}$, dann ist $\mathbf{PO} \neq \mathbf{NPO}$.

Im Laufe dieses Kapitels wird die Struktur der Klasse **NPO** genauer untersucht. Im folgenden liegen daher alle behandelten Optimierungsprobleme in **NPO**.

Bei Optimierungsaufgaben gibt man sich häufig mit Näherungen an die optimale Lösung zufrieden, insbesondere dann, wenn diese „leicht“ zu berechnen sind und vom Wert der optimalen Lösung nicht zu sehr abweichen. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es für ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, kein Verfahren, das eine optimale Lösung in polynomieller Laufzeit ermittelt. Gerade diese Probleme sind in der Praxis jedoch häufig von großem Interesse.

Für eine Instanz $x \in \Sigma_\Pi^*$ und für eine zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ bezeichnet
$D(x,y) = \left| m_\Pi^*(x) - m_\Pi(x,y) \right|$
den **absoluten Fehler von** *y* **bezüglich** *x*.

Ein Algorithmus **A** ist ein **Approximationsalgorithmus (Näherungsalgorithmus)** für Π, wenn er bei Eingabe von $x \in \Sigma_\Pi^*$ eine zulässige Lösung liefert, d.h. wenn $\mathbf{A}(x) \in \mathrm{SOL}_\Pi(x)$ gilt. **A** heißt **absoluter Approximationsalgorithmus (absoluter Näherungsalgorithmus)**, wenn es eine Konstante k gibt mit $D(x, \mathbf{A}(x)) = \left| m_\Pi^*(x) - m_\Pi(x, \mathbf{A}(x)) \right| \leq k$. Das Optimierungsproblem Π heißt in diesem Fall **absolut approximierbar**.

Ist Π ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, so sucht man natürlich nach absoluten Approximationsalgorithmen für Π, die polynomielle Laufzeit aufweisen und für die der Wert $D(x, \mathbf{A}(x))$ möglichst klein ist. Das folgende Beispiel zeigt jedoch, daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht jedes derartige Problem absolut approximierbar ist. Dazu werde der folgende Spezialfall des 0/1-Rucksack-Maximierungsproblems betrachtet:

Das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Instanz: 1. $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{N}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{N}$ das Gewicht und $p_i \in \mathbf{N}$ den Wert (Profit) des Objekts a_i.

$size(I) = n$

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$

3. $m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \mathrm{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß dieses Optimierungsproblem unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ keinen polynomiell zeitbeschränkten Lösungsalgorithmus (der eine *optimale* Lösung ermittelt) besitzt. Es gilt sogar:

Es sei k eine vorgegebene Konstante. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es keinen polynomiell zeitbeschränkten Approximationsalgorithmus **A** für das ganzzahlige 0/1-Rucksack-Maximierungsproblem, der bei Eingabe einer Instanz $I = (A, M)$ eine zulässige Lösung $\mathbf{A}(I) \in \mathrm{SOL}(I)$ berechnet, für deren absoluter Fehler $D(I, \mathbf{A}(I)) = |m^*(I) - m(I, \mathbf{A}(I))| \le k$ gilt.

Die Forderung nach der Garantie der Einhaltung eines absoluten Fehlers ist also häufig zu stark. Es bietet sich daher an, einen Approximationsalgorithmus nach seiner relativen Approximationsgüte zu beurteilen.

Es sei ein Π wieder ein Optimierungsproblem und **A** ein Approximationsalgorithmus für Π (siehe oben), der eine zulässige Lösung $\mathbf{A}(x)$ ermittelt. Ist $x \in \Sigma_\Pi^*$ eine Instanz von Π, so gilt

trivialerweise $\mathbf{A}(x) \le m_{\Pi}^{*}(x)$ bei einem Maximierungsproblem bzw. $m_{\Pi}^{*}(x) \le \mathbf{A}(x)$ bei einem Minimierungsproblem.

Die **relative Approximationsgüte** $R_{\mathbf{A}}(x)$ von **A** bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^{*}$ wird definiert durch

$R_{\mathbf{A}}(x) = \dfrac{m_{\Pi}^{*}(x)}{\mathbf{A}(x)}$ bei einem Maximierungsproblem

bzw.

$R_{\mathbf{A}}(x) = \dfrac{\mathbf{A}(x)}{m_{\Pi}^{*}(x)}$ bei einem Minimierungsproblem.

Bemerkung: Um nicht zwischen Maximierungs- und Minimierungsproblem in der Definition unterscheiden zu müssen, kann man die relative Approximationsgüte von **A** bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^{*}$ auch durch

$$R_{\mathbf{A}}(x) = \max\left\{\frac{m_{\Pi}^{*}(x)}{\mathbf{A}(x)}, \frac{\mathbf{A}(x)}{m_{\Pi}^{*}(x)}\right\}$$

definieren.

Offensichtlich gilt immer $1 \le R_{\mathbf{A}}(x)$. Je dichter $R_{\mathbf{A}}(x)$ bei 1 liegt, um so besser ist die Approximation.

Die Aussage „$R_{\mathbf{A}}(x) \le c$“ mit einer Konstanten $c \ge 1$ für alle Instanzen $x \in \Sigma_{\Pi}^{*}$ bedeutet bei einem Maximierungsproblem $1 \le \dfrac{m_{\Pi}^{*}(x)}{\mathbf{A}(x)} \le c$, also $\mathbf{A}(x) \ge 1/c \cdot m_{\Pi}^{*}(x)$, d.h. der Approximationsalgorithmus liefert eine Lösung, die sich mindestens um $1/c$ dem Optimum nähert. Gewünscht ist also ein möglichst großer Wert von $1/c$ bzw. ein Wert von *c*, der möglichst klein (d.h. dicht bei 1) ist.

Bei einem Minimierungsproblem impliziert die Aussage „$R_{\mathbf{A}}(x) \le c$“ die Beziehung $\mathbf{A}(x) \le c \cdot m_{\Pi}^{*}(x)$, d.h. der Wert der Approximation überschreitet das Optimum um höchstens das *c*-fache. Auch hier ist also ein Wert von *c* gewünscht, der möglichst klein (d.h. dicht bei 1) ist.

4.1 Relativ approximierbare Probleme

Es sei $r \ge 1$. Der Approximationsalgorithmus **A** für das Optimierungsproblem Π aus **NPO** heißt ***r*-approximativer Algorithmus**, wenn $R_{\mathbf{A}}(x) \le r$ für jede Instanz $x \in \Sigma_{\Pi}^{*}$ gilt.

Bemerkung: Es sei **A** ein Approximationsalgorithmus für das Minimierungsproblem Π, und es gelte $\mathbf{A}(x) \le r \cdot m^*_\Pi(x) + k$ für alle Instanzen $x \in \Sigma^*_\Pi$ (mit Konstanten *r* und *k*). Dann ist **A** lediglich $(r+k)$-approximativ und nicht etwa *r*-approximativ, jedoch **asymptotisch** *r*-approximativ (siehe Kapitel 4.3).

Die **Klasse APX** besteht aus denjenigen Optimierungsproblemen aus **NPO**, für die es einen *r*-approximativen Algorithmus für ein $r \ge 1$ gibt.

Offensichtlich ist $\mathbf{APX} \subseteq \mathbf{NPO}$.

Das folgende Optimierungsproblem liegt in **APX**:

Binpacking-Minimierungsproblem:

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \le 1$ für *i* = 1, ..., *n*
   size(*I*) = *n*
2. $\mathrm{SOL}(I) = \left\{ [B_1, \ldots, B_k] \mid [B_1, \ldots, B_k] \text{ ist eine Partition (disjunkte Zerlegung) von } I \text{ mit } \sum_{a_i \in B_j} a_i \le 1 \text{ für } j = 1, \ldots, k \right\}$,
   d.h. in einer zulässigen Lösung werden die Objekte so auf *k* „Behälter“ der Höhe 1 verteilt, daß kein Behälter „überläuft“.
3. Für $[B_1, \ldots, B_k] \in \mathrm{SOL}(I)$ ist $m(I, [B_1, \ldots, B_k]) = k$, d.h. als Zielfunktion wird die Anzahl der benötigten Behälter definiert
4. *Goal* = min

Lösung: Eine Partition der Objekte in möglichst wenige Teile $B_1, \ldots, B_{k^*}$ und die Anzahl k^* der benötigten Teile.

Bemerkung: Da das Laufzeitverhalten der folgenden Approximationsalgorithmen nicht von den Größen der in den Instanzen vorkommenden Objekte abhängt, sondern nur von deren Anzahl, kann man für eine Eingabeinstanz $I = [a_1, \ldots, a_n]$ als Größe den Wert *size*(*I*) = *n* wählen.

Das Binpacking-Minimierungsproblem ist eines der am besten untersuchten Minimierungsprobleme einschließlich der Verallgemeinerungen auf mehrdimensionale Objekte. Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß unter der Voraussetzung $\mathbf{P} \ne \mathbf{NP}$ kein polynomiell zeitbeschränkter Optimierungsalgorithmus erwartet werden kann.

Der folgende in Pseudocode formulierte Algorithmus approximiert eine optimale Lösung:

Nextfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: $B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} nach B_j (in den Behälter mit dem höchsten Index), falls dadurch B_j nicht überläuft, d.h. $\sum_{a_l \in B_j} a_l \leq 1$ gilt; andernfalls lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{NF}(I) =$ Anzahl benötigter nichtleerer Behälter und $B_1, \ldots, B_{\mathbf{NF}(I)}$.

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $\mathbf{NF}(I)/m^*(I) \leq 2$, d.h. der Nextfit-Algorithmus ist 2-approximativ ($\mathbf{NF}(I) \leq 2 \cdot m^*(I)$). Das Binpacking -Minimierungsproblem liegt in **APX**.

Die Grenze 2 im Nextfit-Algorithmus ist asymptotisch optimal: es gibt Instanzen *I*, für die $\mathbf{NF}(I)/m^*(I)$ beliebig dicht an 2 herankommt.

Es gilt sogar eine genauere Abschätzung der relativen Approximationsgüte: Mit $a_{\max} = \max\{a_i \mid i = 1, \ldots, n\}$ ist

$$\mathbf{NF}(I) \leq \begin{cases} (1 + a_{\max}/(1 - a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \leq 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \leq 1 \end{cases}$$

Zu beachten ist, daß der Nextfit-Algorithmus ein online-Algorithmus ist, d.h. die Objekte der Reihenfolge nach inspiziert, und sofort eine Entscheidung trifft, in welchen Behälter ein Objekt zu legen ist, ohne alle Objekte gesehen zu haben. Die Laufzeit des Nextfit-Algorithmus bei einer Eingabeinstanz der Größe *n* liegt in $O(n)$.

Eine asymptotische Verbesserung der Approximation liefert der Firstfit-Algorithmus, der ebenfalls ein online-Algorithmus ist und bei geeigneter Implementierung ein Laufzeitverhalten der Ordnung $O(n \cdot \log(n))$ hat:

Firstfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: $B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls es einen derartigen Behälter unter $B_1, \ldots, B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FF}(I) =$ Anzahl benötigter nichtleerer Behälter und $B_1, \ldots, B_{\mathbf{FF}(I)}$.

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $\mathbf{FF}(I) \leq 1{,}7 \cdot m^*(I) + 2$.

Die Grenze 1,7 im Firstfit-Algorithmus ist ebenfalls asymptotisch optimal: es gibt Instanzen *I*, für die $\mathbf{FF}(I)/m^*(I)$ beliebig dicht an 1,7 herankommt. Zu beachten ist weiterhin, daß der Firstfit-Algorithmus kein 1,7-approximativer Algorithmus ist, sondern nur asymptotisch 1,7-approximativ ist.

Eine weitere asymptotische Verbesserung erhält man, indem man die Objekte vor der Aufteilung auf Behälter nach absteigender Größe sortiert. Die entstehenden Approximationsalgorithmen sind dann jedoch offline-Algorithmen, da zunächst alle Objekte vor der Aufteilung bekannt sein müssen.

FirstfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls es einen derartigen Behälter unter $B_1, \ldots, B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FFD}(I) =$ Anzahl benötigter nichtleerer Behälter und $B_1, \ldots, B_{\mathbf{FFD}(I)}$.

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $\mathbf{FFD}(I) \leq 1{,}5 \cdot m^*(I) + 1$.

BestfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$, der den kleinsten freien Platz aufweist. Falls a_{i+1} in keinen der Behälter $B_1, \ldots, B_j$ paßt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{BFD}(I) =$ Anzahl benötigter nichtleerer Behälter und $B_1, \ldots, B_{\mathbf{BFD}(I)}$.

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $\mathbf{BFD}(I) \le 11/9 \cdot m^*(I) + 4$.

Die folgende Zusammenstellung zeigt noch einmal die mit den verschiedenen Approximationsalgorithmen zu erzielenden Approximationsgüten. In der letzten Zeile ist dabei ein Algorithmus erwähnt, der in einem gewissen Sinne (siehe Kapitel 4.2) unter allen approximativen Algorithmen mit polynomieller Laufzeit eine optimale relative Approximationsgüte erzielt. Zu beachten ist ferner, daß im Sinne der Definition die Algorithmen Firstfit, FirstfitDecreasing und BestfitDecreasing nicht *r*-approximativ (mit $r = 1{,}7$ bzw. $r = 1{,}5$ bzw. $r = 1{,}22$), sondern nur asymptotisch *r*-approximativ sind.

Approximationsalgorithmus	Approximationsgüte
Nextfit	$\mathbf{NF}(I) \le \begin{cases} (1 + a_{\max}/(1 - a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \le 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \le 1 \end{cases}$
Firstfit	$\mathbf{FF}(I) \le 1{,}7 \cdot m^*(I) + 2$
FirstfitDecreasing	$\mathbf{FFD}(I) \le 1{,}5 \cdot m^*(I) + 1$
BestFitDecreasing	$\mathbf{BFD}(I) \le 11/9 \cdot m^*(I) + 4$; $\quad 11/9 = 1{,}222$
Simchi-Levi, 1994	$\mathbf{SL}(I) \le 1{,}5 \cdot m^*(I)$

In Kapitel 4 wurde gezeigt, daß das 0/1-Rucksack-Maximierungsproblem unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ nicht absolut approximierbar ist. Es gibt jedoch für dieses Problem einen 2-approximativen Algorithmus, der aus einer Modifikation der Greedy-Methode des allgemeinen Rucksack-Maximierungsproblems entsteht:

Das 0/1-Rucksackproblem als Maximierungsproblem
(maximum 0/1 knapsack problem)

Instanz: 1. $I = (A, M)$

$A = \{a_1, \ldots, a_n\}$ ist eine Menge von *n* Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, \ldots, n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i.

$size(I) = n$

2. $\mathrm{SOL}(I) = \left\{ (x_1, \ldots, x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, \ldots, n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$; man beachte, daß hier nur Werte $x_i = 0$ oder $x_i = 1$ zulässig sind

3. $m(I,(x_1,...,x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1,...,x_n) \in \mathrm{SOL}(I)$
4. *goal* = max

Lösung: Eine Folge $x_1^*,...,x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1,...,n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I,(x_1^*,...,x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1,...,x_n$, die (1) und (2) erfüllen.

Wendet man die in Kapitel 2.2 beschriebene Greedymethode (Prozedur `greedy_rucksack`) auf eine Instanz *I* an, so liefert diese unter Umständen einen beliebig schlechten Wert der Zielfunktion. Dabei ist natürlich zu beachten, daß innerhalb des Ablaufs der Prozedur `greedy_rucksack` Objekte entweder ganz oder gar nicht in den (teilweise gefüllten) Rucksack gelegt werden, da ja die Bedingung $x_i = 0$ oder $x_i = 1$ für $i = 1,...,n$ einzuhalten ist. Es gilt genauer:

Es bezeichne **GR**(*I*) den Wert der Zielfunktion, den die in Kapitel 2.2 beschriebene Greedymethode bei Eingabe einer Instanz *I* des obigen Problems ermittelt, und $R_{\mathbf{GR}}(I)$ die dabei entstehende relative Approximationsgüte. Dann gibt es für jedes $M \in \mathbf{N}$ eine Instanz *I* mit $R_{\mathbf{GR}}(I) \ge m^*(I)/\mathbf{GR}(I) = (M-1)/3$.

Durch eine leichte Modifikation der Greedymethode kommt man jedoch zu einem Approximationsalgorithmus (d.h. das Problem liegt in **APX**), der hier nur informell beschrieben wird:

1. Bei Eingabe einer Instanz $I = (A,M)$ sortiere man die Objekte nach absteigenden Werten p_i/w_i und wende die in Kapitel 2.2 beschriebene Greedymethode an, wobei innerhalb des Ablaufs der Prozedur `greedy_rucksack` ein Objekt entweder ganz oder gar nicht in den (teilweise gefüllten) Rucksack gelegt wird und dann zum nächsten Objekt übergegangen wird. Der dabei entstehende Wert der Zielfunktion sei **GR**(*I*)
2. Es sei $p_{\max} = \max\{p_i \mid i = 1,...,n\}$. Man vergleiche das Ergebnis im Schritt 1 mit der Rucksackfüllung, die man erhält, wenn man nur das Objekt mit dem Gewinn $p_{\max}$ allein in den Rucksack legt. Man nehme diejenige Rucksackfüllung, die den größeren Wert der Zielfunktion liefert. Dieser werde mit **GRmod**(*I*) bezeichnet; $R_{\mathbf{GRmod}}(I)$ sei die dabei entstehende relative Approximationsgüte.

$$1 \le R_{\mathbf{GRmod}}(I) = m^*(I)/\mathbf{GRmod}(I) \le 2$$

4.2 Grenzen der relativen Approximierbarkeit

Der folgende Satz zeigt, daß es unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht für jedes Optimierungsproblem aus **NPO** einen approximativen Algorithmus gibt. Dazu sei noch einmal das Handlungsreisenden-Minimierungsproblem mit ungerichteten Graphen angegeben:

Das Handlungsreisenden-Minimierungsproblem

Instanz: 1. $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter ungerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

2. $\text{SOL}(G) = \left\{ T \mid T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle \text{ ist eine Tour durch } G \right\}$

3. für $T \in \text{SOL}(G)$, $T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$

4. $goal = \min$

Lösung: Eine Tour $T^* \in \text{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

Ist $\mathbf{P} \neq \mathbf{NP}$, so gibt es keinen r-approximativen Algorithmus für das Handlungsreisenden-Minimierungsproblem (mit $r \in \mathbf{R}_{\geq 1}$).

Ist $\mathbf{P} \neq \mathbf{NP}$, so ist $\mathbf{APX} \subset \mathbf{NPO}$.

Das **metrische Handlungsreisenden-Minimierungsproblem** liegt jedoch in **APX**. Dieses stellt eine zusätzliche Bedingung an die Gewichtsfunktion einer Eingabeinstanz, nämlich die Gültigkeit der **Dreiecksungleichung**:

Für $v_i \in V$, $v_j \in V$ und $v_k \in V$ gilt $w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$.

Auch hierbei ist das zugehörige Entscheidungsproblem **NP**-vollständig. Es gibt (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) im Gegensatz zum (allgemeinen) Handlungsreisenden-Minimierungsproblem für dieses Problem einen 1,5-approximativen Algorithmus (siehe Literatur). Es ist nicht bekannt, ob es einen Approximationsalgorithmus mit einer kleineren relativen Ap-

proximatiosgüte gibt oder ob aus der Existenz eines derartigen Algorithmus bereits $\mathbf{P} = \mathbf{NP}$ folgt.

Hat man für ein Optimierungsproblem aus **APX** einen *r*-approximativen Algorithmus gefunden, so stellt sich die Frage, ob dieser noch verbessert werden kann, d.h. ob es einen *t*-approximativen Algorithmus mit $1 \le t < r$ gibt. Der folgende Satz (gap theorem) besagt, daß man unter Umständen an Grenzen stößt, daß es nämlich Optimierungsprobleme in **APX** gibt, die in polynomieller Zeit nicht beliebig dicht approximiert werden können, außer es gilt $\mathbf{P} = \mathbf{NP}$.

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Minimierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f : \Sigma'^* \to \Sigma^*$ und $c : \Sigma'^* \to \mathbf{N}$ und eine Konstante $e > 0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1+e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus mit $r < 1+e$, außer $\mathbf{P} = \mathbf{NP}$.

Ein entsprechender Satz gilt für Maximierungsprobleme:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Maximierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f : \Sigma'^* \to \Sigma^*$ und $c : \Sigma'^* \to \mathbf{N}$ und eine Konstante $e > 0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1-e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus mit $r < 1/(1-e)$, außer $\mathbf{P} = \mathbf{NP}$.

Mit Hilfe dieser Sätze läßt sich zeigen beispielsweise, daß die Grenze *r* = 1,5 für einen *r*-approximativen (polynomiell zeitbeschränkten) Algorithmus unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ für das Binpacking-Minimierungsproblem optimal ist:

Ist $\mathbf{P} \neq \mathbf{NP}$, dann gibt es keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \le 3/2 - e$ für $e > 0$.

4.3 Polynomiell zeitbeschränkte und asymptotische Approximationsschemata

In vielen praktischen Anwendungen möchte man die relative Approximationsgüte verbessern. Dabei ist man sogar bereit, längere Laufzeiten der Approximationsalgorithmen in Kauf zu nehmen, solange sie noch polynomielles Laufzeitverhalten bezüglich der Größe der Eingabeinstanzen haben. Bezüglich der relativen Approximationsgüte *r* akzeptiert man eventuell ein Laufzeitverhalten, das von $1/(r-1)$ abhängt: Je besser die Approximation ist, um so größer ist die Laufzeit. In vielen Fällen kann man so eine optimale Lösung beliebig dicht approximieren, allerdings zum Preis eines dramatischen Anstiegs der Rechenzeit.

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **polynomiell zeitbeschränktes Approximationsschema** (polynomial-time approximation scheme) für Π, wenn er für jede Eingabeinstanz *x* von Π und für jede rationale Zahl $r > 1$ bei Eingabe von (x,r) eine *r*-approximative Lösung für *x* in einer Laufzeit liefert, die polynomiell von *size*(*x*) abhängt.

Partitionen-Minimierungsproblem

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$
   size(*I*) = *n*
2. $\mathrm{SOL}(I) = \{[Y_1, Y_2] \mid [Y_1, Y_2] \text{ ist eine Partition (disjunkte Zerlegung) von } I\}$
3. Für $[Y_1, Y_2] \in \mathrm{SOL}(I)$ ist $m(I, [Y_1, Y_2]) = \max\left\{\sum_{a_i \in Y_1} a_i, \sum_{a_j \in Y_2} a_j\right\}$
4. *goal* = min

Lösung: Eine Partition der Objekte in zwei Teile $[Y_1, Y_2]$, so daß sich die Summen der Objekte in beiden Teilen möglichst wenig unterscheiden.

Der folgende in Pseudocode formulierte Algorithmus ist ein polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem.

Polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem

Eingabe: $I = [a_1, \ldots, a_n]$, rationale Zahl $r > 1$,
$a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$

Verfahren:

```
VAR Y1 : SET OF INTEGER;
    Y2 : SET OF INTEGER;
    k  : REAL;
    j  : INTEGER;

BEGIN
  IF r >= 2
   THEN BEGIN
          Y1 := {a1,...,an};
          Y2 := { };
        END
   ELSE BEGIN
          Sortiere die Objekte nach absteigender Größe; die dabei
          entstehende Folge sei (x1,...,xn);
          k := ⌈(2−r)/(r−1)⌉;
          finde eine optimale Partition [Y1,Y2] für [x1,...,xk];
          FOR j := k+1 TO n DO
            IF ∑xi∈Y1 xi <= ∑xi∈Y2 xi
            THEN Y1 := Y1 ∪ {xj}
            ELSE Y2 := Y2 ∪ {xj}
        END;
```

Ausgabe: $[Y_1, Y_2]$.

O.B.d.A. sei $\sum_{a_i \in Y_1} a_i \geq \sum_{a_j \in Y_2} a_j$. Man kann zeigen, daß dann $\frac{\sum_{a_i \in Y_1} a_i}{m^*(I)} \leq r$ gilt, d.h. daß der Algorithmus *r*-approximativ ist. Mit $k(r) = \lceil (2-r)/(r-1) \rceil$ ist das Laufzeitverhalten von der Ordnung $O\left(n \cdot \log(n) + n^{k(r)}\right)$. Da $k(r) \in O(1/(r-1))$ ist das Laufzeitverhalten bei festem *r* polynomiell in der Größe *n* der Eingabeinstanz, jedoch exponentiell in *n* und $1/(r-1)$.

Mit **PTAS** werde die Klasse der Optimierungsprobleme in **NPO** bezeichnet, für die es ein polynomiell zeitbeschränktes Approximationsschema gibt.

Offensichtlich ist $\mathbf{PTAS} \subseteq \mathbf{APX}$. In Kapitel 4.2 wurde erwähnt, daß es unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - e$ für $e > 0$ gibt. Daraus folgt:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

Es gibt in **PTAS** Optimierungsprobleme, die ein polynomiell zeitbeschränktes Approximationsschema **A** zulassen, das bei Eingabe einer Instanz x von Π und einer rationalen Zahl $r > 1$ eine r-approximative Lösung für x in einer Laufzeit liefert, die polynomiell von $|x|$ und $1/(r-1)$ abhängt.
Einen derartigen Algorithmus nennt man **voll polynomiell zeitbeschränktes Approximationsschema** (fully polynomial-time approximation scheme). Die Optimierungsprobleme, die einen derartigen Algorithmus zulassen, bilden die Klasse **FPTAS**.

In der Literatur werden Beispiele für Optimierungsprobleme genannt, die in **FPTAS** liegen.

Es läßt sich zeigen:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

In Kapitel 4.1 wurden mehrere Approximationsalgorithmen für das Binpacking-Minimierungsproblem angegeben:

Approximationsalgorithmus	Approximationsgüte
Nextfit	$\mathbf{NF}(I) \leq \begin{cases} (1 + a_{\max}/(1-a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \leq 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \leq 1 \end{cases}$
Firstfit	$\mathbf{FF}(I) \leq 1{,}7 \cdot m^*(I) + 2$
FirstfitDecreasing	$\mathbf{FFD}(I) \leq 1{,}5 \cdot m^*(I) + 1$
BestFitDecreasing	$\mathbf{BFD}(I) \leq 11/9 \cdot m^*(I) + 4;$ $\quad 11/9 = 1{,}222$
Simchi-Levi, 1994	$\mathbf{SL}(I) \leq 1{,}5 \cdot m^*(I)$

Die Approximationsgüte von BestfitDecreasing zeigt, daß man (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) durchaus einen Approximationsalgorithmus entwerfen kann, dessen relative Approximationsgüte unterhalb der überhaupt für einen r-approximativen Algorithmus möglichen Untergrenze liegt. Eventuell gibt es sogar in einem erweiterten Sinne ein polynomiell zeitbeschränktes Approximationsschema für das Binpacking-Minimierungsproblem und andere Optimierungsprobleme. Diese Überlegung führt auf folgende Definition:

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **asymptotisches Approximationsschema** für Π, wenn es eine Konstante k gibt, so daß gilt:
für jede Eingabeinstanz x von Π und für jede rationale Zahl $r > 1$ liefert **A** bei Eingabe von (x,r) eine (zulässige) Lösung, deren relative Approximationsgüte $R_{\mathbf{A}}(x)$ die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m^*_{\Pi}(x)$ erfüllt. Außerdem ist die Laufzeit von **A** für jedes feste r polynomiell in der Größe *size*(x) der Eingabeinstanz.

Zur Erinnerung: die relative Approximationsgüte wurde definiert durch

$R_{\mathbf{A}}(x) = \dfrac{m^*_{\Pi}(x)}{\mathbf{A}(x)}$ bei einem Maximierungsproblem

bzw.

$R_{\mathbf{A}}(x) = \dfrac{\mathbf{A}(x)}{m^*_{\Pi}(x)}$ bei einem Minimierungsproblem.

Die Bedingung $R_{\mathbf{A}}(x) \le r + k/m^*_{\Pi}(x)$ besagt also

bei einem Maximierungsproblem: $\mathbf{A}(x) \ge 1/r \cdot m^*_{\Pi}(x) - k'$ mit $k' = k/\left(r + k/m^*_{\Pi}(x)\right) \le k/r$,

daher $\mathbf{A}(x) \ge 1/r \cdot m^*_{\Pi}(x) - k/r$

bzw.

bei einem Minimierungsproblem: $\mathbf{A}(x) \le r \cdot m^*_{\Pi}(x) + k$.

Die Bezeichnung „asymptotisches Approximationsschema“ ist aus der Tatsache zu erklären, daß für „große“ Eingabeinstanzen *x* der Wert $m^*_{\Pi}(x)$ der Zielfunktion einer optimalen Lösung ebenfalls groß ist. Daher gilt in diesem Fall $\lim_{size(x)\to\infty} R_{\mathbf{A}}(x) \le r$.

Die Klasse aller Optimierungsprobleme, die ein asymptotisches Approximationsschema zulassen, wird mit $\mathbf{PTAS}^{\infty}$ bezeichnet.

Das Binpacking-Minimierungsproblem liegt in $\mathbf{PTAS}^{\infty}$, d.h. es kann asymptotisch beliebig dicht in polynomieller Zeit approximiert werden (auch wenn $\mathbf{P} \neq \mathbf{NP}$ gilt).

Es gilt sogar: Es gibt ein asymptotisches Approximationsschema, das polynomiell in der Problemgröße und in $1/(r-1)$ ist.

Die für das Binpacking-Minimierungsproblem eingesetzten Approximationsschemata arbeiten offline, da die Objekte einer Eingabeinstanz zunächst nach absteigender Größe sortiert werden. In Kapitel 5.1 wird ein online-Approximationsschema beschrieben, dessen relative Approximationsgüte im Mittel beliebig dicht an das Optimum kommt.

Die Klasse $\mathbf{PTAS}^{\infty}$ ordnet sich in die übrigen Approximationsklassen ein:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{PTAS}^{\infty} \subset \mathbf{APX} \subset \mathbf{NPO}$.

5 Weiterführende Konzepte

Die Analyse des Laufzeitverhaltens eines Algorithmus **A** im schlechtesten Fall $T_{\mathbf{A}}(n) = \max\{t_{\mathbf{A}}(x) \mid x \in \Sigma^* \text{ und } |x| \le n\}$ liefert eine *Garantie* (obere Schranke) für die Zeit, die er bei einer Eingabe zur Lösung benötigt[4]. Für jede Eingabe $x \in \Sigma^*$ mit $|x| = n$ gilt $t_{\mathbf{A}}(x) \le T_{\mathbf{A}}(n)$. Dieses Verhalten ist oft jedoch nicht charakteristisch für das Verhalten bei „den meisten" Eingaben. Ist eine Wahrscheinlichkeitsverteilung P der Eingabewerte bekannt oder werden alle Eingaben als gleichwahrscheinlich aufgefaßt, so kann man das Laufzeitverhalten von **A** untersuchen, wie es sich im Mittel darstellt. Die **Zeitkomplexität** von **A** im **Mittel** wird definiert als

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}[t_{\mathbf{A}}(x) \mid |x| = n] = \int_{|x|=n} t_{\mathbf{A}}(x) \cdot dP(x).$$

Interessiert bei der Analyse des Algorithmus **A** weniger sein Laufzeitverhalten, sondern beispielsweise die „Qualität" seiner Ausgabe, etwa die Güte der Approximation bei einem Näherungsverfahren, so kann auch hier durch Erwartungswertbildung über das Gütemaß eine Analyse im Mittel durchgeführt werden. Auch hier werden verschiedene Ansätze verfolgt, die in den folgenden dargestellt werden.

5.1 Mittleres Verhalten von Algorithmen bei bekannter Verteilung der Eingabeinstanzen

Es sei **A** ein Approximationsalgorithmus zur näherungsweisen Lösung eines Optimierungsproblems Π über Σ^*. Für eine Eingabeinstanz $x \in \Sigma^*$ bezeichne $\mathbf{A}(x)$ den von **A** ermittelten Wert der Zielfunktion und $m_{\Pi}^*(x)$ den Wert einer optimalen Lösung. Dann interessieren beispielsweise folgende Erwartungswerte:

- erwartetes Approximationsergebnis bei Eingaben der Größe n: $\mathbf{E}[\mathbf{A}(x) \mid |x| = n]$
- erwartete relative Approximationsgüte bei Eingaben der Größe n: $\mathbf{E}[m_{\Pi}^*(x)/\mathbf{A}(x) \mid |x| = n]$ bzw. $\mathbf{E}[\mathbf{A}(x)/m_{\Pi}^*(x) \mid |x| = n]$
- erwartete Approximationsgüte bei „großen" bzw. fast allen Eingabegrößen: $\lim_{n\to\infty} \mathbf{E}[\mathbf{A}(x) \mid |x| = n]$

[4] $t_{\mathbf{A}}(x) =$ Anzahl der Anweisungen, die von **A** zur Berechnung von $\mathbf{A}(x)$ durchlaufen werden.

- erwartete relative Approximationsgüte bei „großen“ bzw. fast allen Eingabengrößen: $\lim_{n\to\infty} \mathbf{E}\left[m_{\Pi}^{*}(x)/\mathbf{A}(x)\,|\,|x|=n\right]$ bzw. $\lim_{n\to\infty} \mathbf{E}\left[\mathbf{A}(x)/m_{\Pi}^{*}(x)\,|\,|x|=n\right]$.

Beispiel:

Es sei $I_n=[a_1,\ldots,a_n]$ eine Eingabeinstanz für das Binpacking-Minimierungsproblem (siehe Kapitel 4.1). Es werde angenommen, daß die Objekte über dem Intervall $]0,1]$ gleichverteilt sind. $\mathbf{NF}(I_n)$ bezeichne die Anzahl der Behälter, die der Nextfit-Algorithmus zur Packung von I_n benötigt. Dann läßt sich zeigen:
$\mathbf{E}\left[\mathbf{NF}(I_n)\right]=2/3\cdot n$ und $\mathbf{E}\left[\mathbf{NF}(I_n)/m^{*}(I_n)\right]=2n/(3\cdot 1/2\cdot n)=4/3$; die erwartete „Füllhöhe“ eines Behälters ist $3/4$, und im Mittel enthält jeder Behälter 1,5 viele Objekte.

Beim Entwurf eines Approximationsalgorithmus für ein Optimierungsproblem kann man u.U. in Kauf nehmen, daß er sich bei einigen Eingaben sehr schlecht verhält, im Mittel aber ein gutes Approximationsverhalten aufweist. Man kann also versuchen, bei Kenntnis der Verteilung der möglichen Eingaben im Algorithmenentwurf diese Verteilung zu berücksichtigen, um einen günstigen Wert beispielsweise für die erwartete relative Approximationsgüte $\lim_{n\to\infty} \mathbf{E}\left[m_{\Pi}^{*}(x)/\mathbf{A}(x)\,|\,|x|=n\right]$ bzw. $\lim_{n\to\infty} \mathbf{E}\left[\mathbf{A}(x)/m_{\Pi}^{*}(x)\,|\,|x|=n\right]$ bei „großen“ bzw. fast allen Eingabengrößen zu erzielen.

Beispiel:

Der folgende Algorithmus (in Pseudocode formuliert) ist ein polynomiell zeitbeschränktes Approximationsschema für das Binpacking-Minimierungsproblem, das online arbeitet, d.h. jedes Objekt nur einmal inspiziert und eine Packungsentscheidung trifft, ohne die nachfolgenden Objekte zu kennen.

Online Approximationsschema für das Binpacking-Minimierungsproblem (online Faltungsalgorithmus)

Eingabe: $I=[a_1,\ldots,a_n]$, $s\in\mathbf{N}$ mit $s\geq 4$ und $s\equiv 0 \bmod(2)$
$a_1,\ldots,a_n$ („Objekte“) sind rationale Zahlen mit $0<a_i\leq 1$ für i = 1, ..., n

Verfahren:

```
VAR idx : INTEGER;
    k   : INTEGER;
    I   : ARRAY [1..s] OF Intervall ⊆ ]0,1];

BEGIN
  FOR idx := 1 TO s DO
```

```
  I[idx] := ⌋ (s−idx)/s , (s−idx+1)/s ⌊ ;

FOR k := 1 TO n DO
  BEGIN
    idx := s + 1 - ⌈s·a_k⌉; { Bestimmung eines Intervalls
                              I[idx] mit a_k ∈ I[idx]     }
    IF (idx >= 2)
       AND
       (es gibt einen Behälter B, der ein Objekt b ∈ I[s−(idx - 2)]
        enthält)
    THEN BEGIN
           B := B ∪ {a_k};
            kennzeichne B als gefüllt;
         END
    ELSE BEGIN
           plaziere a_k in einen neuen Behälter;
           { für idx = 1 ist dieser Behälter gefüllt }
         END;
  END;
```

Ausgabe: $\mathbf{OFD}(I) =$ Anzahl benötigter nichtleerer Behälter und $B_1, ..., B_{\mathbf{OFD}(I)}$.

Es läßt sich zeigen:

Sind die Objekte in einer Eingabeinstanz $I = [a_1, \ldots, a_n]$ für das Binpacking-Minimierungsproblem im Intervall $]0,1]$ gleichverteilt, so gilt $\lim_{n\to\infty} \mathbf{E}\left[m^*(I_n)/\mathbf{OFD}(I_n)\right] \geq 1 - 1/(s+1)$.

Das Approximationsschema ist genau auf die Voraussetzung der Gleichverteilung der Objekt in einer Eingabeinstanz zugeschnitten. Daher kann es vorkommen, daß es sich bei nicht Erfüllung dieser Voraussetzung „schlecht“ verhält. Die Idee des obigen Approximationsschemas läßt sich jedoch auf allgemeinere Verteilungen als der Gleichverteilung der Objekte im Intervall $]0,1]$ erweitern, nämlich auf stetige Verteilungen, deren Dichtefunktion monoton fällt.

5.2 Randomisierte Algorithmen

Im ansonsten deterministischen Algorithmus werden als zulässige Elementaroperationen Zufallsexperimente zugelassen. Ein derartiges Zufallsexperiment kann beispielsweise mit Hilfe eines Zufallszahlengenerators ausgeführt werden. So wird beispielsweise auf diese Weise entschieden, in welchem Teil einer Programmverzweigung der Algorithmus während seines

Ablaufs fortgesetzt wird. Andere Möglichkeiten zum Einsatz eines Zufallsexperiments bestehen bei Entscheidungen zur Auswahl möglicher Elemente, die im weiteren Ablauf des Algorithmus als nächstes untersucht werden sollen.

Man nennt derartige Algorithmen **randomisierte Algorithmen.**

Grundsätzlich gibt es zwei Klassen randomisierter Algorithmen: **Las-Vegas-Verfahren**, die stets – wie von deterministischen Algorithmen gewohnt – ein korrektes Ergebnis berechnen. Daneben gibt es **Monte-Carlo-Verfahren**, die ein korrektes Ergebnis nur mit einer gewissen Fehlerwahrscheinlichkeit, aber in jedem Fall effizient, bestimmen. Häufig findet man bei randomisierten Algorithmen einen Trade-off zwischen Korrektheit und Effizienz.

Beispiele für randomisierte Algorithmen vom Las-Vegas-Typ sind:

- `Quicksort`, wie er in Kapitel 2.1 beschrieben wird: die Position des Elements für die Aufteilung der unsortierten Element in zwei Teile wird zufällig bestimmt. Das Laufzeitverhalten zur Sortierung von *n* Elementen liegt daher im Mittel und in nahezu allen praktischen Fällen in $O(n \cdot \log(n))$
- Einfügen von Primärschlüsselwerten in einen binären Suchbaum: Statt die Schlüssel $S_1, ..., S_n$ in dieser Reihenfolge sequentiell in den binären Suchbaum einzufügen, wird jeweils der nächste einzufügende Schlüssel aus den restlichen, d.h. noch nicht eingefügten Schlüsseln zufällig ausgewählt und in den binären Suchbaum eingefügt. Das Ergebnis ist eine mittlere Baumhöhe der Ordnung $O(\log(n))$

Ein Beispiel für einen randomisierten Algorithmus vom Monte-Carlo-Typ ist der Rabin'sche Primzahltest, der hier informell beschrieben werden soll (Details findet man in der angegebenen Literatur).

Um zu testen, ob eine Zahl *n* eine Primzahl ist oder nicht, kann man für große *n* nicht systematisch alle Zahlen $\leq \sqrt{n}$ als potentielle Teiler von *n* untersuchen, da es zu viele Zahlen sind (exponentiell viele bezogen auf die Stellenzahl von *n*). Man sucht daher bei Vorgabe von *n* nach einem „Zeugen" (witness) für das Zusammengesetzsein von *n*. Ein Zeuge ist dabei eine Zahl *k* mit $1 \leq k \leq n-1$, der eine bestimmt Eigenschaft zukommt, aus der man schließen kann, daß *n* zusammengesetzt ist. Die Eigenschaft „*k* ist ein Teiler von *n*" ist nicht geeignet, da es im allgemeinen zu viele Nicht-Zeugen zwischen 1 und $n-1$ gibt. Die Eigenschaft muß vielmehr so definiert werden, daß gilt: Falls *n* zusammengesetzt ist, kommt mehr als der Hälfte aller Zahlen zwischen 1 und $n-1$ die Eigenschaft zu, d.h. mehr als die Hälfte aller Zahlen zwischen 1 und $n-1$ sind Zeugen für das Zusammengesetzsein von *n*. Derartige Eigenschaften sind seit langem aus der Zahlentheorie bekannt (siehe Literatur), z.B.:

k ist ein Zeuge für das Zusammengesetztsein von *n*, wenn gilt:

$1 \le k \le n-1$ und entweder nicht $k^{n-1} \equiv 1 \pmod{n}$
oder es gibt ein i, so daß 2^i ein Teiler von $n-1$ ist und
$1 < ggT\left(k^{(n-1)/2^i} - 1, n\right) < n$ gilt

Wenn n ungerade und zusammengesetzt ist, dann gibt es mindestens $(n-1)/2$ Zeugen dafür.

Dann läßt sich folgender Primzahltest definieren:

Randomisierter Primzahltest

Eingabe: $n \in \mathbf{N}$, n ist ungerade, $m \in \mathbf{N}$

Verfahren:

```
VAR idx             : INTEGER;
    k               : INTEGER;
    zusammengesetzt : BOOLEAN;

BEGIN
  zusammengesetzt := FALSE;
  FOR idx := 1 TO m DO
    BEGIN
      wähle eine Zufallszahl k zwischen 1 und n-1;
      IF (k ist Zeuge für das Zusammengesetztsein von n)
      THEN BEGIN
             zusammengesetzt := TRUE;
             Break;
           END;
    END;
END;
```

Ausgabe: Antwort: n ist Primzahl, wenn `zusammengesetzt = FALSE` gilt, ansonsten ist n keine Primzahl.

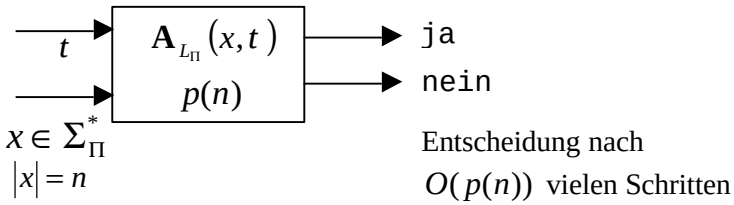
Man sieht leicht, daß dieser Algorithmus eine korrekte Antwort mit einer Wahrscheinlichkeit ermittelt, die mindestens gleich $1-(1/2)^m$ ist .

5.3 Modelle randomisierter Algorithmen

In Zusammenführung der obigen Ansätze mit den Modellen aus der Theorie der Berechenbarkeit (Turing-Maschinen, deterministische und nichtdeterministische Algorithmen) wurde eine Reihe weiterer Berechnungsmodelle entwickelt. Im folgenden werden wieder Entscheidungsprobleme betrachtet.

Ausgehend von der Klasse **P** der deterministisch polynomiell entscheidbaren Probleme erweitert man deren Algorithmen nicht um die Möglichkeit der Verwendung nichtdeterministisch erzeugter Zusatzinformationen (Beweise) wie beim Übergang zu den polynomiellen Verifizierern, sondern läßt zu, daß bei Verzweigungen während des Ablaufs der Algorithmen (Verzweigungen) ein Zufallsexperiment darüber entscheidet, welche Alternative für den weiteren Ablauf gewählt wird. Man gelangt so zu der Klasse **RP** (randomized polynomial time).

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. L_Π liegt in **RP** (bzw. „Π liegt in **RP**“), wenn es einen Algorithmus (Akzeptor) $\mathbf{A}_{L_\Pi}$ gibt, der neben der Eingabe $x \in \Sigma_\Pi^*$ eine Folge $t \in \{0,1\}^*$ zufälliger Bits liest, wobei jedes Bit unabhängig von den vorher gelesenen Bits ist und $P(0) = P(1) = 1/2$ gilt. Nach polynomiell vielen Schritten (in Abgängigkeit von der Größe $|x|$ der Eingabe) kommt der Akzeptor auf die `ja`-Entscheidung bzw. auf die `nein`-Entscheidung. Eingaben für $\mathbf{A}_{L_\Pi}$ sind also die Wörter $x \in \Sigma_\Pi^*$ und eine Folge $t \in \{0,1\}^*$ zufälliger Bits:



Für jedes $x \in L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}) \geq 1/2$,
für jedes $x \notin L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{nein}) = 1$

Man läßt also zur Akzeptanz von $x \in L_\Pi$ einen einseitigen Fehler zu. Jedoch muß bei $x \in L_\Pi$ der Algorithmus $\mathbf{A}_\Pi$ bei mindestens der Hälfte aller möglichen Zufallsfolgen τ auf die `ja`-Entscheidung kommen. Für die übrigen darf er auch auf die `nein`-Entscheidung führen. Für $x \notin L_\Pi$ muß er aber immer die `nein`-Entscheidung treffen. Der einseitige Fehler kann durch Einsatz geeigneter Replikations-Techniken beliebig klein gehalten werden.

Es gilt $\mathbf{P} \subseteq \mathbf{RP}$ und $\mathbf{RP} \subseteq \mathbf{NP}$. Ob diese Inklusionen echt sind, ist nicht bekannt, vieles spricht jedoch dafür.

Die Zusammenführung der Konzepte des Zufalls und des Nichtdeterminismus bei polynomiell zeitbeschränktem Laufzeitverhalten führt auf die Klasse **PCP** (probabilistically check-

able proofs). Hierbei werden Verifizierer, wie sie bei der Definition der Klasse **NP** eingeführt wurden, um die Möglichkeit erweitert, Zufallsexperimente auszuführen:

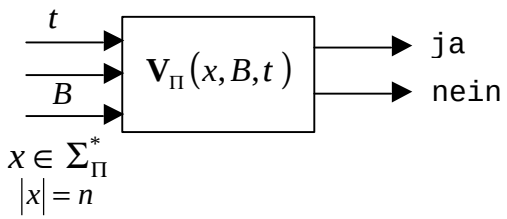
Es seien $r : \mathbf{N} \to \mathbf{N}$ und $q : \mathbf{N} \to \mathbf{N}$ Funktionen. Ein Verifizierer (nichtdeterministischer Algorithmus) $\mathbf{V}_\Pi$ heißt $(r(n), q(n))$**-beschränkter Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π, wenn er Zugriff auf eine Eingabe $x \in \Sigma_\Pi^*$ mit $|x| = n$, einen Beweis $B \in \Sigma_0^*$ und eine Zufallsfolge $t \in \{0,1\}^*$ hat und sich dabei folgendermaßen verhält:

1. $\mathbf{V}_\Pi$ liest zunächst die Eingabe $x \in \Sigma_\Pi^*$ (mit $|x| = n$) und $O(r(n))$ viele Bits aus der Zufallsfolge $t \in \{0,1\}^*$.
2. Aus diesen Informationen berechnet $\mathbf{V}_\Pi$ $O(q(n))$ viele Positionen der Zeichen von $B \in \Sigma_0^*$, die überhaupt gelesen (erfragt) werden sollen.
3. In Abhängigkeit von den gelesenen Zeichen in B (und der Eingabe x) kommt $\mathbf{V}_\Pi$ auf die `ja`- bzw- `nein`-Entscheidung.

Die Entscheidung, die $\mathbf{V}_\Pi$ bei Eingabe von $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$ und $t \in \{0,1\}^*$ liefert, wird mit $\mathbf{V}_\Pi(x, B, t)$ bezeichnet.

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. $L_\Pi \in \mathbf{PCP}(r(n), q(n))$ **(„**Π liegt in der Klasse $\mathbf{PCP}(r(n), q(n))$**“)**, wenn es einen $(r(n), q(n))$-beschränkten Verifizierer gibt, der L_Π in folgender Weise akzeptiert:

Für jedes $x \in L_\Pi$ gibt es einen Beweis B_x mit $P(\mathbf{V}_\Pi(x, B_x, t) = \texttt{ja}) = 1$,
für jedes $x \notin L_\Pi$ und alle Beweise B gilt $P(\mathbf{V}_\Pi(x, B, t) = \texttt{nein}) \geq 1/2$.



Für Eingaben $x \in L_\Pi$ gibt es also einen Beweis, der immer akzeptiert wird. Der Versuch, eine Eingabe $x \notin L_\Pi$ zu akzeptieren, scheitert mindestens mit einer Wahrscheinlichkeit $\geq 1/2$.

Im folgenden bezeichnet $poly(n)$ ein Polynom. Dann gilt beispielsweise

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right) = TIME\left(poly(n)\right) \text{ und } \mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right) = NTIME\left(poly(n)\right).$$

Es gelten folgende Aussagen:

$$\mathbf{PCP}\left(poly(n), poly(n)\right) = NTIME\left(2^{poly(n)}\right),$$

$$\mathbf{PCP}(0,0) = \mathbf{P},$$

$$\mathbf{PCP}\left(0, poly(n)\right) = \mathbf{NP},$$

$$\mathbf{PCP}\left(r(n), q(n)\right) \subseteq NTIME\left(q(n) \cdot 2^{O(r(n))}\right),$$

$$\mathbf{PCP}\left(\log(n), poly(n)\right) = \mathbf{NP}$$

Das folgende Ergebnis (Arora, Lund, Motwani, Sudan, Szegedy, 1992) wird als das wichtigste Resultat der theoretischen Informatik der letzten 10 Jahre angesehen:

$$\mathbf{PCP}\left(\log(n), 1\right) = \mathbf{NP}$$

„Wie man Beweise verifiziert, ohne sie zu lesen“

Das Ergebnis besagt, daß es zu jeder Menge L_Π für ein Entscheidungsproblem Π aus **NP** einen Verifizierer gibt, der bei jeder Eingabe nur konstant viele Stellen des Beweises liest, die er unter Zuhilfenahme von $O\left(\log(n)\right)$ vielen zufälligen Bits auswählt, um mit hoher Wahrscheinlichkeit richtig zu entscheiden.

6 Anhang

Der Anhang stellt einige allgemeine mathematische Grundlagen zusammen.

6.1 Wichtige Funktionen

Die von der Problemgröße abhängige Zeitkomplexität eines Algorithmus ist häufig eine reelwertige Funktion der natürlichen Zahlen. Typische Funktionen, die als Zeitkomplexitäten auftreten, sind Polynome, Logarithmus- und Exponentialfunktionen, Wurzelfunktionen und arithmetische Verknüpfungen dieser Funktionen. In diesem Zusammenhang sind daher einige mathematische Ergebnisse von Interesse, die das Wachstumsverhalten dieser Funktionen beschreiben.

Eine Funktion

$$p: \begin{cases} \mathbf{R} \to \mathbf{R} \\ x \to a_n x^n + a_{n-1} x^{n-1} + \ \ldots \ a_1 x + a_0 \end{cases}$$

mit reellen Konstanten $a_n, a_{n-1}, \ \ldots \ , a_1, a_0$ und $a_n \neq 0$ heißt **Polynom vom Grad *n***. Für $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ \ldots \ + a_1 x + a_0$ schreibt man wie üblich $p(x) = \sum_{i=0}^{n} a_i x^i$.

Es sei $a > 0$ eine reelle Konstante. Die Funktion

$$\exp_a : \begin{cases} \mathbf{R} & \to & \mathbf{R}_{>0} \\ x & \to & a^x \end{cases}$$

heißt **Exponentialfunktion zur Basis *a***.

Jede so definierte Exponentialfunktion $\exp_a$ ist bijektiv und besitzt daher eine Umkehrfunktion $\exp_a^{-1}$, die **Logarithmusfunktion zur Basis *a*** heißt und mit $\log_a$ bezeichnet wird:

$$\log_a : \begin{cases} \mathbf{R}_{>0} & \to & \mathbf{R} \\ x & \to & \log_a(x) \end{cases}$$

Im folgenden wird als Definitionsbereich der auftretenden Funktionen jeweils die Menge der natürlichen Zahlen genommen.

Für die **Exponentialfunktion** zur Basis $a > 1$ $\exp_a : \begin{cases} \mathbf{N} & \to & \mathbf{R}_{>0} \\ n & \to & a^n \end{cases}$ gilt:

$\lim_{n\to\infty} a^n = \infty$ und $\lim_{n\to-\infty} a^n = 0$,

$\lim_{n\to\infty} a^{-n} = 0$ und $\lim_{n\to-\infty} a^{-n} = \infty$.

Für die **Logarithmusfunktion** zur Basis $a > 1$ $\log_a : \begin{cases} \mathbf{N} & \to & \mathbf{R} \\ n & \to & \log_a(n) \end{cases}$ gilt:

$\lim_{n\to\infty} \log_a(n) = \infty$.

Exponentialfunktionen wachsen sehr schnell. Es gilt

$a^{n+1} = a \cdot a^n$,

d.h. bei Vergrößerung des Argumentwerts um 1 vergrößert sich der Funktionswert um den Faktor *a*.

Logarithmusfunktionen wachsen sehr langsam. Es gilt

$\lim_{n\to\infty} \left(\log_a(n+1) - \log_a(n)\right) = 0$,

d.h. obwohl die Logarithmusfunktion bei wachsendem Argumentwert gegen ∞ strebt, nehmen die Funktionswerte letztlich nur noch geringfügig zu.

Über das Wachstumsverhalten der Exponential- bzw. Logarithmusfunktionen zur Basis $a > 1$ verglichen mit Polynomen gibt die folgende Zusammenstellung Auskunft:

(a) Es sei $p(x)$ ein Polynom. Dann gilt

$$\lim_{n\to\infty} \frac{|p(n)|}{a^n} = 0.$$

Exponentialfunktionen wachsen also schneller als jedes Polynom.

(b) Für jedes $m \in \mathbf{N}$ ist

$$\lim_{n\to\infty} \left(\frac{(\log_a(n))^m}{n} \right) = 0.$$

Man sieht, daß selbst Potenzen von Logarithmusfunktionen im Verhältnis zu Polynomen (sogar zu Polynomen 1. Grades) langsam wachsen.

(c) Für jedes $m \in \mathbf{N}$ ist

$$\lim_{n\to\infty} \left(\frac{\log_a(n)}{\sqrt[m]{n}} \right) = 0.$$

Man sieht, daß Logarithmusfunktionen im Verhältnis zu Wurzelfunktionen langsam wachsen.

Die folgende Tabelle zeigt fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$ und einige ausgewählte (gerundete) Funktionswerte.

Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5
i	$h_i(n)$	$h_i(10)$	$h_i(100)$	$h_i(1000)$
1	$\log_2(n)$	3,3219	6,6439	9,9658
2	$\sqrt{n}$	3,1623	10	31,6228
3	n	10	100	1000
4	n^2	100	10.000	1.000.000
5	2^n	1024	$1{,}2676506 \cdot 10^{30}$	$> 10^{693}$

Die folgende Tabelle zeigt noch einmal die fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$. Es sei $y_0 > 0$ ein fester Wert. Die dritte Spalte zeigt für jede der fünf Funktionen Werte n_i mit $h_i(n_i) = y_0$. In der vierten Spalte sind diejenigen Werte $\overline{n_i}$ aufgeführt, für die $h_i(\overline{n_i}) = 10 \cdot y_0$ gilt, d.h. dort ist angegeben, auf welchen Wert man n_i vergrößern muß, damit der Funktionswert auf den 10-fachen Wert wächst. Wie man sieht, muß bei der Logartihmusfunktion wegen ihres langsamen Wachstums der Wert stark vergrößert werden, während bei der schnell anwachsenden Exponentialfunktion nur eine additive konstante Steigerung um ca. 3,3 erforderlich ist.

Spalte 1	Spalte 2	Spalte 3	Spalte 4
i	$h_i(n)$	n_i mit $h_i(n_i) = y_0$	$\overline{n_i}$ mit $h_i(\overline{n_i}) = 10 \cdot y_0$
1	$\log_2 n$	n_1	$(n_1)^{10}$
2	$\sqrt{n}$	n_2	$100 \cdot n_2$
3	n	n_3	$10 \cdot n_3$
4	n^2	n_4	$\approx 3{,}162 \cdot n_4$
5	2^n	n_5	$\approx n_5 + 3{,}322$

Die Logarithmusfunktion zu einer Basis $B > 1$ gibt u.a. näherungsweise an, wieviele Ziffern benötigt werden, um eine natürliche Zahl im Zahlensystem zur Basis *B* darzustellen:

Gegeben sei die Zahl $n \in \mathbf{N}$ mit $n > 0$. Sie benötige $m = m(n, B)$ signifikante Stellen zur Darstellung im Zahlensystem zur Basis *B*, d.h.

$n = \sum_{i=0}^{m-1} a_i \cdot B^i$ mit $a_i \in \{0,\ 1,\ \ldots,\ B-1\}$ und $a_{m-1} \neq 0$.

Es ist $B^{m-1} \le n < B^m$ und folglich $m-1 \le \log_B(n) < m$. Daraus ergibt sich für die Anzahl der benötigten Stellen, um eine Zahl *n* im Zahlensystem zur Basis *B* darzustellen, $m(n,B) = \lfloor \log_B(n) \rfloor + 1 = \lceil \log_B(n+1) \rceil$.

6.2 Größenordnung von Funktionen

Im folgenden seien $f : \mathbf{N} \to \mathbf{R}$ und $g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Die Funktion *f* ist von der (**Größen-) Ordnung** $O(g(n))$, geschrieben $f(n) \in O(g(n))$, wenn gilt:
Es gibt eine Konstante $c > 0$, die von *n* nicht abhängt, so daß $|f(n)| \le c \cdot |g(n)|$ für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen ist.

Einige Regeln:

$f(n) \in O(f(n))$

Für *d* = const. ist $d \cdot f(n) \in O(f(n))$

$O(O(f(n))) = O(f(n))$

$O(f(n) \cdot g(n)) = O(f(n)) \cdot O(g(n))$

$O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$

$n^m \in O(n^k)$ für $m \le k$

$O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$

Ist *p* ein Polynom vom Grade *m*, so ist $p(n) \in O(n^k)$ für $k \ge m$

Konvergiert $f(n) = \sum_{i=0}^{\infty} a_i \cdot n^i$ für $n \le r$, so ist für jedes $k \ge 0$: $f(n) = \sum_{i=0}^{k} a_i \cdot n^i + g(n)$ mit $g(n) \in O(n^{k+1})$.

$\log_a(n) \in O(\log_b(n))$, denn $\log_a(n) = \frac{1}{\log_b(a)} \cdot \log_b(n)$

$e^{h(n)} \in O(2^{O(h(n))})$, denn $e^{h(n)} = 2^{\log_2(e) \cdot h(n)}$

Für jedes Polynom $p(n)$ und jede Exponentialfunktion a^n mit $a>1$ ist $a^n \notin O(p(n))$, jedoch $p(n) \in O(a^n)$

Für jede Logarithmusfunktion $\log_a(n)$ mit $a>1$ und jedes Polynom $p(n)$ ist $p(n) \notin O(\log_a(n))$, jedoch $\log_a(n) \in O(p(n))$

Für jede Logarithmusfunktion $\log_a(n)$ mit $a>1$ und jede Wurzelfunktion $\sqrt[m]{n}$ ist $\sqrt[m]{n} \notin O(\log_a(n))$, jedoch $\log_a(n) \in O(\sqrt[m]{n})$

Es seien $a>1$, $b>1$, $m \in \mathbf{N}_{>0}$, $p(n)$ ein Polynom; dann ist $O(\log_b(n)) \subset O(\sqrt[m]{n}) \subset O(p(n)) \subset O(a^n)$

Es seien $f: \mathbf{N} \to \mathbf{R}$ und $g: \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Es wird definiert:
$f(n) \in \Omega(g(n))$ („Groß Omega“), wenn gilt:
Es gibt eine Konstante $c>0$, die von n nicht abhängt, so daß $|f(n)| \geq c \cdot |g(n)|$ für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen ist.

Es gilt $f(n) \in O(g(n))$ genau dann, wenn $g(n) \in \Omega(f(n))$ ist.

Gilt sowohl $f(n) \in O(g(n))$ als auch $f(n) \in \Omega(g(n))$, dann ist $f(n) \in \Theta(g(n))$ („Groß Theta“). In diesem Fall gibt es also zwei von n unabhängige Konstanten c_1 und c_2 mit $c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|$ für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen.

Literaturauswahl

Die folgende Literaturauswahl wurde im vorliegenden Texte verwendet. Im Text sind die entsprechenden Stellen jedoch nicht explizit gekennzeichnet. Insbesondere Beweise zu den einzelnen Aussagen und Sachverhalten sind den folgenden Quellen zu entnehmen.

Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

Blum, N.: **Theoretische Informatik**, Oldenbourg, 1998.

Bovet, D.P.; Crescenzi, P.: **Introduction to the Theory of Complexity**, Prentice Hall, 1994.

Dewdney, A.K.: **Der Turing Omnibus**, Springer, 1995.

Garey, M.R.; Johnson, D.: **Computers and Intractability, A Guide to the Theory of NP-Completeness**, Freeman, San Francisco, 1979.

Harel, D.: **Algorithmics**, 2nd Ed., Addison Wesley, 1992.

Herrmann, D.: **Algorithmen Arbeitsbuch**, Addison-Wesley, 1992.

Hoffmann, U.: A class of simple online stochastic bin packing algorithms, Computing 29, 227-239, 1982.

Hoffmann, U.: **Stochastische Packungsalgorithmen**, Dissertation, Stuttgart, 1981.

Ottmann, T. (Hrsg.): **Prinzipien des Algorithmenentwurfs**, Spektrum Akademischer Verlag, 1998.

Ottmann, T.; Widmayer, P.: **Algorithmen und Datenstrukturen**, 3. Aufl., Spektrum Akademischer Verlag, 1996.

Turau, V.: **Algorithmische Graphentheorie**, Addison Wesley, 1996.

Wegener, I. (Hrsg.): **Highlights aus der Informatik**, Springer, 1996.