Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2022

Prof. Dr. Steffen Reith
Steffen.Reith@hs-rm.de

Hochschule RheinMain
Fachbereich Design Informatik Medien

Erstellt von:	Steffen Reith
Zuletzt überarbeitet von:	Steffen Reith
Email:	steffen.reith@hs-rm.de
Erste Version vollendet:	Juli 2006
Version:	7878be9
Date:	2022-04-19 10:22:05 +0200

I conclude that there are two ways of
constructing a software design:
One way is to make it so simple that
there are obviously no deficiencies and the
other way is to make it so complicated that
there are no obvious deficiencies.

C.A.R. Hoare

Als ich im ersten Semester war, war mein
Professor so dumm, dass ich ihn kaum ertragen
konnte. Aber als ich ins fünfte Semester
kam, war ich doch erstaunt, wieviel der alte
Mann in vier Semestern dazugelernt hatte.

frei nach Mark Twain

Dieses Skript ist aus der Vorlesung „Algorithmen und Datenstrukturen“ (früher Informatik 1) des Bachelor- und Diplom-Studiengangs „Allgemeine Informatik“ an der Fachhochschule Wiesbaden hervorgegangen und wurde mit LaTeX gesetzt. Ich danke allen Hörern dieser Vorlesung für konstruktive Anmerkungen und Verbesserungen. Besonders hervorzuheben sind hier Herr Robert Augustin (Abschnitt 3.2.3), Herr Frank Meffert (Abbildung 27), Herr Arne Zastrow (Abschnitt 2.1.2) und Herr Patrick Vogt, die Verbesserungen beigesteuert und Tippfehler angemerkt haben. Naturgemäß ist ein Skript nie fehlerfrei (ganz im Gegenteil!) und es ändert (mit Sicherheit!) sich im Laufe der Zeit. Deshalb bin ich auf weitere Verbesserungvorschläge angewiesen und freue mich auch über weitere Anregungen durch die Hörer.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1. Algorithmen und Probleme

1.1. Was ist ein Algorithmus?

Definition „Algorithmus“ (informell):

> Ein Algorithmus ist eine eindeutige (Berechnungs-) Vorschrift die beschreibt, wie man Eingabeobjekte in Ausgabeobjekte umwandelt (vgl. z.B. [Bre95]).

Beispiel 1 (Kochen von Eiern nach P. Bocuse 1977): *Wir gehen wie folgt vor:*

1. *In einem ausreichend großen Topf Wasser zum Kochen bringen*
2. *Das Ei in ein Sieb mit groben Löchern legen*
3. *Das Sieb in das kochende Wasser senken*
4. *Wenn das Ei mittelgroß ist, dann* 9 *Minuten kochen*
5. *Wenn das Ei groß ist, dann* 10 *Minuten kochen*
6. *Sieb aus dem kochenden Wasser nehmen*
7. *Sieb in kaltes Wasser tauchen*

Beispiel 2 (Berechnen des ggTs nach Euklid ca. 300 v.Chr.): *Der größter gemeinsamer Teiler (kurz: ggT) der Zahlen a und b wird wie folgt bestimmt:*

1. *Wenn a größer als b, dann tausche beide Zahlen*
2. *Teile b (ganzzahlig) durch a mit Rest r*
3. *Setze b auf den Wert von a und a auf den Wert von r*
4. *Ist r größer als* 0*, arbeite bei Schritt* 2 *weiter*
5. *Gib b aus (dies ist der ggT)*

Zusätzlich wollen wir fordern, dass diese Vorschrift (= Algorithmus) von einem mechanischen oder elektronischen Gerät, dem Prozessor, ausgeführt werden kann (womit Beispiel 1 unserem Verständnis eines Algorithmus nicht entspricht). Algorithmen sollen also (Berechnungs-) Probleme lösen. Um solche Probleme formal besser darstellen zu können, benutzen wir immer folgende Notation:

Problem:	EIERKOCHEN
Eingabe:	Ein Ei
Ausgabe:	Ein hartgekochtes Ei

oder

Problem:	GGT
Eingabe:	Zwei natürliche Zahlen a und b
Ausgabe:	Der größte gemeinsame Teiler von a und b

Eine konkrete Eingabe eines Problems wollen wir auch *Instanz* nennen. Für lösbare Probleme gibt es beliebig viele Lösungsalgorithmen, wobei wir den für uns optimalen[1] Algorithmus suchen.

[1] Es ist klar, dass der Begriff des „optimalen Algorithmus“ vom Einsatzzweck abhängt.

Ein Algorithmus muss den folgenden Anforderungen genügen:

- **Endlichkeit** - Der Algorithmus muss eine *endliche Beschreibung* (= endliche Anzahl von „Befehlen“) haben.
- **Determiniertheit** - Jeder Eingabewert in den Algorithmus führt zu einem eindeutigen Resultat. Verarbeitet der Algorithmus diesen Wert erneut, so muss er das gleiche Ergebnis liefern.
- **Vollständigkeit** - Ein Algorithmus *hält* für jede zulässige Eingabe nach endlich vielen Schritten an (= terminieren).
- **Universalität** - Alle (zulässigen) Eingabedaten werden korrekt verarbeitet.
- **Nachvollziehbarkeit** - Ein Algorithmus muss von Dritten nachvollzogen / überprüft werden können.

Erfüllt der Algorithmus von Bocuse diese Eigenschaften?

- **Endlichkeit** - Offensichtlich „ja“, da die Beschreibung aus sieben Zeilen / Anweisungen besteht.
- **Determiniertheit** - „Jaein“, da nicht klar ist, ob jedes große Ei wirklich hart gekocht wird.
- **Vollständigkeit** - „Jaein“, da nicht klar ist, ob das Programm hält, denn das Ei bleibt ja im kalten Wasser liegen (Terminierung fraglich).
- **Universalität** - „Nein“, da das Programm nicht mit kleinen Eier umgehen kann (evtl. kleine Eier nicht zu den erlaubten Eingaben zählen!).
- **Nachvollziehbarkeit** - „Jaein“, da nicht jedem klar ist, was ein „ausreichend großer Topf“, „ein Sieb mit groben Löchern“ oder ein „mittelgroßes Ei“ ist.

Die meisten „Algorithmen des täglichen Lebens“ genügen unseren Anforderungen nicht, denn weder entsprechen sie den obigen Kriterien (vgl. Montageanleitung eines großen schwedischen Möbelhauses), noch bestehen sie aus (elementaren) Anweisungen, die ein mechanischer / elektronischer Prozessor verarbeiten kann.

Wir benötigen eine (formale) eindeutige Notation für Algorithmen

Bemerkung 3: *Abschließend noch einige weiterführende Bemerkungen:*

- *Den Algorithmenbegriff kann man exakter fassen, indem man einen „abstrakten Computer“ (Stichwort: Berechnungsmodell) definiert. Ein bekanntes Beispiel ist die „Turing-Maschine“, die nach* Alan M. Turing[2] *benannt wurde. Hier sind die zulässigen Anweisungen mathematisch exakt definiert und man legt fest: „Ein Algorithmus ist eine endliche Folge von Befehlen einer Turing-Maschine“.*

[2] *1912 in London (England) - †1954 in Wilmslow (England)

- *Aufgrund der „*Church'schen These*“ (nach* Alonso Church[3]*) ist man heute überzeugt, dass der Algorithmus- / Berechenbarkeitsbegriff von Turing-Maschinen äquivalent zu jeder hinreichend umfangreichen (modernen) Programmiersprache ist. Dies bedeutet, dass wir der Einfachkeit halber C, JAVA, oder eine C-ähnliche Notation (z.B. Pseudocode) benutzen können, um unsere Algorithmen zu beschreiben.*

- *Die „Church'sche These“, „Äquivalenz von Berechenbarkeitsbegriffen“ und „die Grenzen von Algorithmen“ werden in der Rekursionstheorie bzw. Komplexitätstheorie studiert, die Teilgebiete der Theoretischen Informatik darstellen.*

1.2. Notationen für Algorithmen

Die derzeit verwendeten „halbformalen“ Mittel zur Beschreibung von Algorithmen benutzen die Grundelemente, die man in den meisten (höheren) Programmiersprachen findet. Dabei gibt es an C angelehnte verbale Beschreibungen (oft „Pseudocode“ genannt) und graphische Beschreibungen (Nassi-Shneiderman Diagramme oder Programmablaufpläne). Wir legen fest: Jeder Algorithmus soll nur mit Hilfe der folgenden Bausteine konstruiert werden (Stichwort: „strukturierte Programmierung“).

1.2.1. Sequenz

Die einzelnen Teilalgorithmen werden hintereinander in der Reihenfolge ausgeführt, in der sie aufgeschrieben sind:

Anweisung1;
Anweisung2;
Anweisung3;

Das entsprechende Nassi-Shneiderman Diagramm zeigt Abbildung 1(b).

1.2.2. Verzweigung

Wenn die Bedingung B erfüllt ist, so wird der „ja-Teilalgorithmus“ ausgeführt, sonst (also wenn die Bedingung nicht erfüllt ist) der „nein-Teilalgorithmus“. Es werden *nie* beide Teilalgorithmen ausgeführt:

if *(B)* **then**
  ja-Teilalgorithmus;
else
  nein-Teilalgorithmus;
end

Das entsprechende Nassi-Shneiderman Diagramm zeigt Abbildung 1(c).

1.2.3. Fallunterscheidung

Eine Spezialform der Verzweigung ist die Fallunterscheidung. Wenn die Auswertung des Ausdrucks A den Wert i ergibt, dann führen wir Teilalgorithmus i aus. Wenn kein passender Teilalgorithmus i existiert, dann wird der Teilalgorithmus „default“ ausgeführt:

[3] *1903 in Washington (USA) - †1995 in Hudson (USA)

```
switch (A) do
    case 1: do
        Teilalgorithmus1;
        break;
    case 2: do
        Teilalgorithmus2;
        break;
    case 3: do
        Teilalgorithmus1;
        break;
    otherwise do
        Teilalgorithmus „default“;
    end
end
```

Das entsprechende Nassi-Shneiderman Diagramm zeigt Abbildung 1(a).

1.2.4. Abweisende Schleife

Hier wird der Teilalgorithmus S (= Scheifenkörper / Schleifenrumpf) solange ausgeführt, wie die Bedingung B erfüllt ist. Der Name „abweisende Schleife“ kommt daher, dass die Bedingung zu Beginn getestet und der Schleifenrumpf evtl. nicht ausgeführt wird:

```
while (B) do
    Teilalgorithmus S;
end
```

Das entsprechende Nassi-Shneiderman Diagramm zeigt Abbildung 1(d).

1.2.5. Nichtabweisende Schleife

Hier wird erst der Teilalgorithmus S (= Scheifenkörper / Schleifenrumpf) ausgeführt und dann die Bedingung B geprüft. Damit ist auch die Bezeichnung „nichtabweisende Schleife“ plausibel, denn der Teilalgorithmus S wird auf jeden Fall einmal ausgeführt:

```
repeat
    Teilalgorithmus S;
until (B);
```

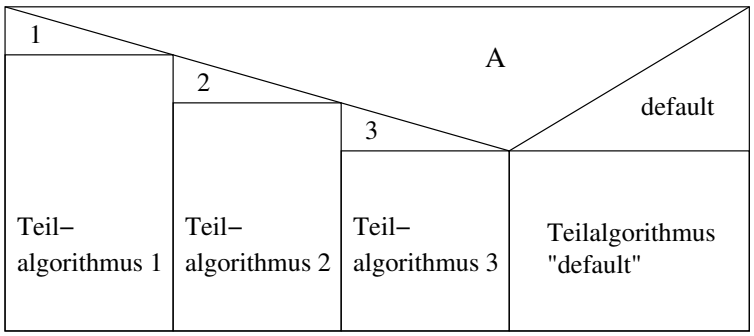
Das entsprechende Nassi-Shneiderman Diagramm zeigt Abbildung 1(e).

1.2.6. Zählschleife

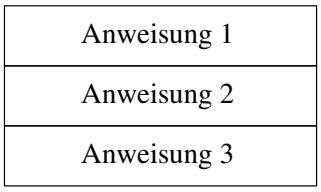
Bei der Zählschleife wird der Schleifenindex i vom Startwert hin zum Endwert inkrementiert. Dabei wird der Teilalgorithmus S (= Scheifenkörper) jedesmal durchlaufen. Der Unterschied zur abweisenden und nichtabweisende Schleife ist, dass schon zu Beginn die Anzahl der Schleifendurchläufe fest steht:

```
for (i = Start to Ende) do
    Teilalgorithmus S;
end
```

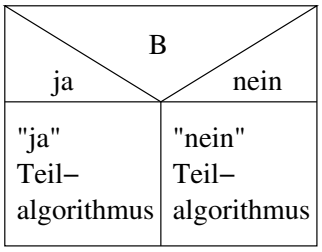
Das entsprechende Nassi-Shneiderman Diagramm zeigt 1(f).



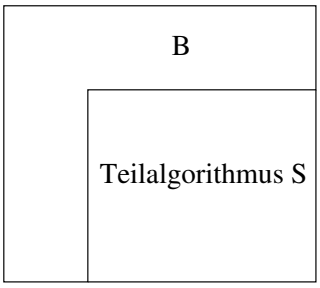
(a) Fallunterscheidung



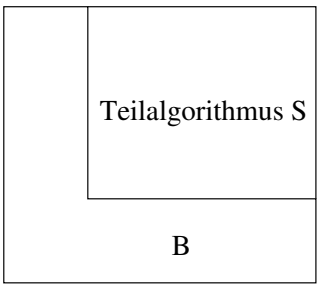
(b) Sequenz



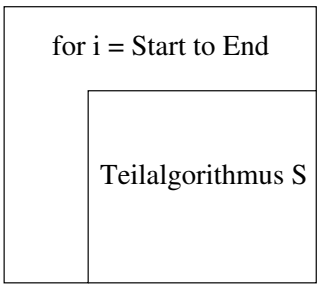
(c) Verzweigung



(d) Abweisende Schleife



(e) Nichtabweisende Schleife



(f) Zählschleife

Abbildung 1: Die grundlegenden Bestandteile von Nassi-Shneiderman Diagrammen

1.2.7. Abschließende Bemerkungen

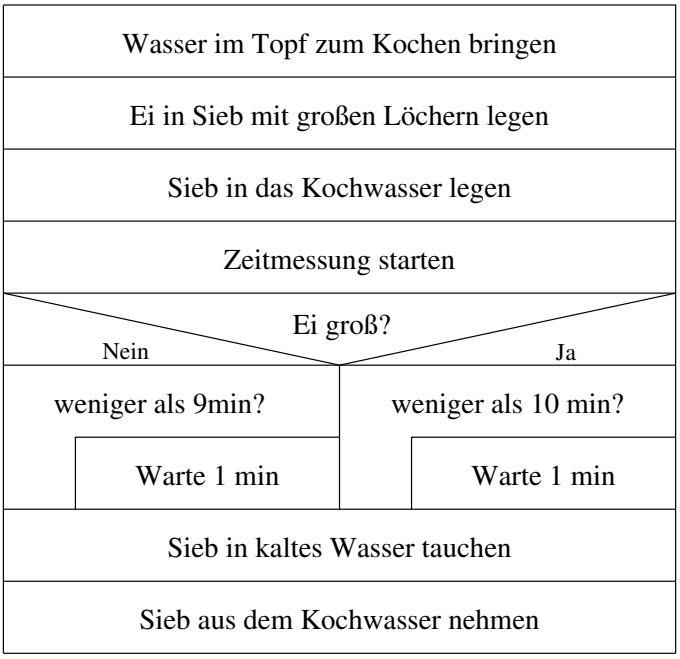
Alle oben vorgestellten Programmbausteine haben ein gemeinsames Merkmal. Jeder dieser Bausteine hat

- genau einen Eingang und
- genau einen Ausgang.

Durch diese Konstruktion wird die Verständlichkeit des Algorithmus sehr erhöht (vgl. „Nachvollziehbarkeit“). Viele Programmiersprachen kennen auch *Sprunganweisungen* (vgl. „GOTO“), die meist zu sehr unübersichtlichen und schwer wartbaren Code führen. Aus diesem Grund verwenden *wir* die Sprunganweisung *nicht*. In der Praxis werden in sehr seltenen Fällen Sprunganweisungen benutzt, um die Verständlichkeit zu verbessern. Solche Fälle werden in dieser Vorlesung nicht auftreten!

1.2.8. Anwendungen der Notationen für Algorithmen

Beispiel 4: *Darstellung des Vorgangs „Eier kochen“ nach Paul Bocuse:*



Die graphische Darstellung mit Nassi-Shneiderman Diagrammen kann bei umfangreichen Algorithmen schnell unübersichtlich werden. Weiterhin wird dann das manuelle Erstellen von solchen Diagrammen sehr aufwändig, weshalb in der Praxis spezielle Tools zum Einsatz kommen sollten. In solchen Fällen ist die Verwendung von Pseudocode oft einfacher, wie das folgende Beispiel zeigt:

Beispiel 5: *Der Euklidische Algorithmus in Pseudocode:*

Eingabe: *Zwei Zahlen a und b*
Ergebnis: *Der größte gemeinsame Teiler von a und b*

```
begin
    if (a > b) then
        tausche a und b;
    end
    repeat
                                          /* Ganzzahlige Division (optional) */
        q = b/a;
                                                    /* Berechne den Rest */
        r = b%a;
        b = a;
        a = r;
    until (r > 0);
    return b;
end
```

Algorithmus 1: Euklidischer Algorithmus

Bemerkung 6: *Neben Nassi-Shneiderman Diagrammen und Pseudocode-Notation sind so genannte Programmablaufpläne bekannt, die heute aber aufgrund ihrer Unübersichtlichkeit kaum noch eingesetzt werden.*

1.2.9. Die Korrektheit von Algorithmen

Wir haben u.a. die „Vollständigkeit“ und „Universalität“ als Anforderung an Algorithmen gefordert. Diese Anforderungen wollen wir kurz als die *Korrektheit* eines Algorithmus bezeichnen. Wir werden sehen, dass der (formale) Beweis für die Korrektheit eines Algorithmus sehr aufwändig und kompliziert sein kann.

Beispiel 7: *Als Beispiel soll die Formulierung des Euklidische Algorithmus aus Beispiel 2 dienen:*

- *Liefert der Algorithmus für alle Eingaben ein (korrektes) Ergebnis?*
  Antwort: NEIN
  - *Wähle b beliebig und $a = 0$, dann ist das Ergebnis undefiniert, da die Division durch 0 nicht erlaubt ist.*
  - *Was passiert, wenn die Eingaben keine ganzen Zahlen sind?*

  Abhilfe: Sonderfälle abfragen (z.B. $\mathrm{ggT}(0,0) = 0$ und $\mathrm{ggT}(a,0) = \mathrm{ggT}(0,a) = a$) und Eingaben auf positive ganze Zahlen einschränken (vgl. Formulierung in Abschnitt 1.2.8).
- *Terminiert der Algorithmus mit diesen Einschränkungen?*
  Antwort: JA
  Beim Teilen von b durch a mit Rest r gilt:

  $$b = q \cdot a + r, \text{wobei } 0 \leq r < a.$$

  Damit wird in jedem Schleifendurchlauf der Wert der Variablen r kleiner. Zusätzlich kann sie nicht kleiner 0 werden, d.h. der Algorithmus muss nach maximal a vielen Schritten terminieren.

- *Liefert der Algorithmus für Eingaben $a, b \neq 0$ immer das richtige Ergebnis?*
  Antwort: JA
  Sei d der größte gemeinsame Teiler von a und b:
  - *Vor der Schleife gilt die Eigenschaft $d = \mathrm{ggT}(a, b)$ immer noch, denn das evtl. Vertauschen der Argumente ändert nichts am* ggT *(*Vorbedingung*).*
  - *Am Ende der Schleife ist die Eigenschaft weiterhin gültig, denn*

$$\begin{array}{rrcl} & b & = & q \cdot a + r \\ \Leftrightarrow & d \cdot b' & = & q \cdot d \cdot a' + r \\ \Leftrightarrow & d \cdot (b - q \cdot a') & = & r \end{array}$$

    D.h. d ist ein Teiler von r und damit gilt: $\mathrm{ggT}(b, a) = \mathrm{ggT}(a, \overbrace{b\%a}^{=r})$*. Damit ändert sich die* ggT*-Eigenschaft nicht (*Schleifeninvariante*).*
  - *Nach dem Abbruch der Schleife wird dann der größte gemeinsame Teiler zurückgegeben, denn* $\mathrm{ggT}(b, 0) = b$ *(*Nachbedingung*).*

Bemerkung 8:

- *Bei der Analyse von Algorithmen ist es oft schwer, die „Vorbedingung“ und die „Schleifeninvariante“ zu finden, die dann einen einfachen Korrektheitsbeweis ermöglichen.*
- *Hat man „Vorbedingung“ und „Schleifeninvariante“ gefunden, so kann man verhältnismäßig einfach einen Induktionsbeweis über die Korrektheit führen („Vorbedingung“ $\triangleq$ „Induktionsanfang“ und die „Schleifeninvariante“ liefert den „Induktionsschritt“).*
- *Durch Testen kann die Korrektheit eines Algorithmus / Programms nicht gezeigt werden.*

2. Suchen und Sortieren

2.1. Suchen in Feldern

Eine wichtige Operation, die Bestandteil vieler Berechnungsaufgaben ist, ist das Suchen (in Arrays):

PROBLEM: SEARCH
EINGABE: Feld A, Anzahl der Komponenten n, Schlüssel s
AUSGABE: Index i, so dass $A[i] == s$

Bei dieser Variante des Suchproblems gehen wir davon aus, dass der Schlüssel *immer* in A vorkommt. Man kann dies aber auch modifizieren:

PROBLEM: SEARCH´
EINGABE: Feld A, Anzahl der Komponenten n, Schlüssel s
AUSGABE: *undefiniert*, falls s nicht in A vorkommt, sonst Index i, so dass $A[i] == s$

Mit ein wenig Überlegung wird klar, dass man SEARCH´ leicht in SEARCH überführen kann (vgl. Algorithmus 2). Als mögliche Anwendung des Suchproblems könnte man z.B. an das Suchen in Telefonbüchern denken. Es ist offensichtlich, dass dieses Problem von zentraler Bedeutung für Algorithmen ist und viele praktische Anwendungen hat.

2.1.1. Die sequentielle Suche

Der offensichtlichste Algorithmus zur Lösung des Suchproblems ist, die Komponenten eines Arrays von Index 0 nach Index $n-1$ (oder umgekehrt) zu durchlaufen. Dieses Vorgehen ist bekannt als *sequentielle Suche* (siehe Algorithmus 2).

Eingabe: Feld A (Daten ab Index 1 abgelegt), Anzahl der Komponenten n, Schlüssel s
Ergebnis: Index i, so dass $A[i] == s$ oder 0 falls nicht vorhanden ($0 \triangleq$ „undefiniert“)

```
begin
    /* Auf jeden Fall den Schlüssel finden */
    A[0] = s;
    /* Vom höchsten zum niedrigsten Index suchen */
    i = n;
    while (Teste ob A[i] den Schlüssel s nicht enthält) do
        gehe zur nächsten Komponente;
    end
    return i;
end
```

Algorithmus 2: Sequentielle Suche (löst SEARCH´)

Analyse:

- Wird der Schlüssel nicht gefunden, so brauchen wir $n+1$ Vergleiche.
- Im besten Fall braucht die sequentielle Suche nur einen Vergleich.
- Nehmen wir an, jede Komponente in A wird mit der gleichen Wahrscheinlichkeit gesucht, dann ist die durchschnittliche Anzahl von Vergleichen:

$$\frac{1}{n}(1+2+3+\cdots+n) = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

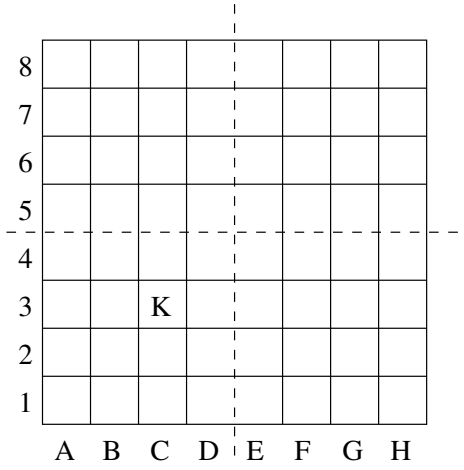
  (genau die Hälfte der Vergleiche wie im erfolglosen Fall!)

2.1.2. Die binäre Suche

Das Suchproblem kann aber auch (wesentlich!) schneller gelöst werden, wenn die Daten sortiert sind (also eine Ordnung haben). Dazu wird eine Technik verwendet, die als „Teile-und-Herrsche“ (engl. „divide-and-conquer“) bekannt ist. Diese Methode ist ein weit verbreitetes „Design-Pattern“ für Algorithmen und benutzt die folgenden Ideen:

- **Divide:** Teile ein Problem in mehrere (oft zwei) Teilprobleme auf.
- **Conquer:** Löse jedes Teilproblem rekursiv, falls es noch groß genug ist. Kleine Teilprobleme werden direkt gelöst.
- **Combine:** Setze die Lösungen für die Teilprobleme zu einer Lösung für das gesamte Problem zusammen.

Beispiel 9: *Finde den König K durch gezieltes Fragen:*



- *Alternative* 1*: Nach* $64 = 8 \cdot 8$ *ist die Position ermittelt (worst case).*
- *Alternative* 2*: Frage* 1*: „Ist der König auf der rechten Seite?“, Antwort: Nein, Frage* 2*: „Ist der König auf der unteren Hälfte?“, Antwort: Ja, ... Nach* 6 *Fragen ist die Position des Königs bekannt. Diese Technik ist unter dem Begriff* „Binäre Suche“ *bekannt.*

Mit Hilfe dieser Idee können wir die *Binäre Suche* in Feldern implementieren (siehe Algorithmus 3).

Beispiel 10: *Gegeben sei folgendes Feld A mit* 18 *Elementen:*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$A[i]$	1	5	9	10	11	17	19	23	27	52	63	65	78	80	90	91	93	99

Wir fragen, ob 65 *in A vorkommt und bestimmen den Index falls dies der Fall ist. Dazu rufen wir die Suchfunktion mit „binsearch(A, 0, 17, 65)“ auf. Die folgende Tabelle gibt an, mit welchen Argumenten die Funktion* `binsearch` *aufgerufen (Rekursion) und welcher Wert für den Index des mittleren Elements ermittelt wird.*

Rekursionstiefe	*min*	*max*	*Schlüssel*	*Mitte*
0	0	17	65	8
1	9	17	65	13
2	9	12	65	10
3	11	12	65	11
4	65 *an Position* 11 *gefunden*			

Bemerkung 11:

- *Wir haben Algorithmus 3 rekursiv formuliert. Es ist einfach (und effizienter) eine nicht rekursive Variante der Binären Suche zu formulieren (Übung).*
- *Um die Laufzeit für n Elemente abzuschätzen, gehen wir o.B.d.A. (≙ ohne Beschränkung der Allgemeinheit) davon aus, dass die Anzahl n eine Zweierpotenz ist, etwa* $n = 2^m$*. Bei jedem rekursiven Aufruf wird die Anzahl der zu durchsuchenden Elemente halbiert (entweder das Intervall* $[\min, \ldots, \text{mitte}[$ *oder das Intervall* $]\text{mitte}, \ldots, \max]$ *wird durchsucht). Damit bricht die Rekursion nach m Schritten ab. Im Allgemeinen brauchen wir also etwa* $\log_2(n)$ *Schritte, um ein Element zu finden.*

Eingabe: Feld A, Index „min“ und Index „max“ die den Bereich der Suche angeben, Schlüssel s
Ergebnis: Index i, so dass $A[i] == s$ oder `undefiniert` wenn der Schlüssel nicht gefunden wurde

```
int binsearch(A, min, max, s)
begin
    if (min <= max) then /* Rekursion noch nicht abbrechen? */
                                    /* Index des mittleren Elements berechnen */
        mitte = ⌊(min + max)/2⌋;
        if (A[mitte] == s) then /* Schlüssel gefunden? */
            return mitte;
        else
            if (s < A[mitte]) then /* Auf linker Seite weitersuchen? */
                 /* auf die linke Seite ohne mittleres Element einschränken */
                max = mitte − 1;
            else
                 /* auf die rechte Seite ohne mittleres Element einschränken */
                min = mitte + 1;
            end
             /* Mit neuem Intervall (linke oder rechte Hälfte) weitersuchen */
            return binsearch(A, min, max, s);
        end
    else
            /* Das Intervall ist ungültig, d.h. Schlüssel nicht gefunden */
        return undefiniert;
    end
end
```

Algorithmus 3: Binäre Suche

2.2. Einfache Sortierverfahren

In diesem Abschnitt sollen Algorithmen für das folgende Problem angegeben werden:

PROBLEM: SORT
EINGABE: Feld A, Anzahl der Komponenten n
AUSGABE: permutiertes Array A', so dass $A'[0] \leq A'[1] \leq A'[2] \leq \cdots \leq A'[n-1]$

2.2.1. Insertion Sort

Wir starten mit einem Verfahren, das als „Insertion Sort“ bekannt ist. Die Idee hierbei ist vergleichbar mit der Technik, mit der viele Leute Spielkarten sortieren:

i) starte mit leerer linker Hand

ii) nehme die nächste Karte vom Tisch

iii) suche die korrekte Position der Karte in der linken Hand (von links nach rechts)

iv) füge Karte an der gefundenen Position ein

v) bearbeite die nächste Karte

Eine Pseudocodedarstellung von Insertion Sort ist in Algorithmus 4 angegeben.

Eingabe: Feld A mit n Komponenten
Ergebnis: Sortiertes Feld A' mit $A'[0] \leq A'[1] \leq A'[2] \leq \cdots \leq A'[n-1]$

```
1  for (j = 1 to n − 1) do /* unsortierte Elemente bearbeiten */
2  |  key = A[j];
3  |  i = j − 1;
4  |  while ((i ≥ 0) && (A[i] > key)) do /* korrekt Einfügen */
5  |  |  A[i + 1] = A[i];
6  |  |  i−−;
7  |  end
8  |  A[i + 1] = key;
9  end
10 return A;
```

Algorithmus 4: Insertion Sort

Beispiel 12: *Wir sortieren die folgenden Daten mit Insertion Sort:*

i	0	1	2	3	4
$A[i]$	3	7	11	1	5

Für $j == 1$ *und* $j == 2$ *ändert sich nichts, weil das Subarray* $3, 7, 11$ *schon sortiert ist.*

Im Fall $j == 3$ *bewirkt die innere Schleife (Zeilen* 4-7*):*

3 7 11 1 5
3 7 11 11 5
3 7 7 11 5
3 3 7 11 5
1 3 7 11 5

1 3 7 11 5

Im Fall $j == 4$ *bewirkt die innere Schleife (Zeilen* 4-7*):*

1 3 7 11 5
1 3 7 11 11
1 3 7 7 11
1 3 5 7 11

1 3 5 7 11

Korrektheit des Insertion Sort Es stellt sich die Frage, warum dieser Algorithmus korrekt ist. Dazu machen wir folgende Beobachtung: Die Komponenten $A[0], \ldots, A[j-1]$ sind zu Beginn der äußeren Schleife sortiert (vgl. linke Hand), die restlichen Komponenten sind noch unsortiert (vgl. Kartenstapel).

Damit ergibt sich die „Schleifeninvariante“:

> Zu Beginn des Durchlaufs des Rumpfes der äußeren Schleife (siehe Zeile 2) besteht das Subarray $A[0, \ldots, j-1]$ aus den Originalkomponenten $A[0], \ldots, A[j-1]$ ($\triangleq$ Eingabe vor dem Programmstart) und $A[0, \ldots, j-1]$ ist sortiert.

Eine Schleifeninvariante ist ein Eigenschaft, die durch einen Schleifendurchlauf nicht geändert wird. Für die Korrektheit des Algorithmus müssen wir die Richtigkeit der Vorbedingung, der Schleifeninvariante und der Nachbedingung zeigen:

- „Vorbedingung“: Wir starten mit $j == 1$, d.h. das sortierte Subarray besteht nur aus der Komponente $A[0]$, die auch mit der Originalkomponente $A[0]$ übereinstimmt und sortiert ist.
- „Korrektheit der Schleifeninvariante“: Der Schleifenkörper der inneren while-Schleife (ab Zeile 4) verschiebt die Elemente nach „rechts“ (zum größeren Index) und platziert $A[j]$ an der korrekten Position im Subarray (es bleibt also sortiert). Dann wird j inkrementiert und die Schleifeninvariante gilt für $j + 1$.
- „Nachbedingung“: Die for-Schleife (Zeile 1) bricht ab, wenn $j == n$ gilt, d.h. das Subarray $A[0, \ldots, n-1]$ enthält alle ursprünglichen Elemente und ist sortiert.

Dies zeigt, dass der Algorithmus korrekt ist.

Laufzeit des Insertion Sorts Bei der Analyse von Algorithmen untersucht man meist den Speicher-, Kommunikations- oder Zeitbedarf. Hier: *Untersuchung des Zeitbedarfs.*

Dazu zählen wir jede „elementare Anweisung“ im Pseudocode (Unterprogrammaufrufe sind nicht elementar!) mit einem Schritt. Wir wollen die Anzahl der Schritte bestimmen, die wir für das Sortieren von n Elementen benötigen. Sei t_j die Ausführungsanzahl der while-Bedingung (Zeile 4) für den Wert j. Die Anzahl der Schritte $T(n)$ beträgt dann:

$$\begin{aligned} T(n) &= n \text{ /* Zeile 1, Schleifendurchläufe: } (n-1), \text{ 1 Abbruch */} \\ &+ (n-1) \text{ /* Zeile 2 */} \\ &+ (n-1) \text{ /* Zeile 3 */} \\ &+ \textstyle\sum_{j=1}^{n-1} t_j \text{ /* Zeile 4, } n-1 \text{ Durchläufe */} \\ &+ \textstyle\sum_{j=1}^{n-1} (t_j - 1) \text{ /* Zeile 5 */} \\ &+ \textstyle\sum_{j=1}^{n-1} (t_j - 1) \text{ /* Zeile 6 */} \\ &+ (n-1) \text{ /* Zeile 8 */} \\ &+ 1 \text{ /* Zeile 10 */} \end{aligned}$$

Im schlimmsten Fall ist das Feld A zu Beginn rückwärts sortiert, d.h. jeder Schleifendurchlauf der while-Schleife (Zeile 4) benötigt $j + 1$ Schritte. Wir wissen:

$$\begin{aligned} \sum_{i=1}^{n-1} j &= \tfrac{n(n-1)}{2} \\ \sum_{i=1}^{n-1} (j+1) &= \tfrac{n(n-1)}{2} + (n-1) \end{aligned}$$

Damit ergibt sich für die Anzahl der ausgeführten Schritte:

$$\begin{aligned} T(n) &= n + 4(n-1) + 3\tfrac{n(n-1)}{2} + 1 \\ &= 5n - 4 + \tfrac{3}{2}n^2 - \tfrac{3}{2}n + 1 \\ &= \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 3 \end{aligned}$$

Bemerkung 13: *Wir haben hier den „schlimmsten Fall“ angenommen (worst-case Analyse). Manchmal betrachtet man auch den „durchschnittlichen Fall“ (average-case Analyse). Solche Analysen können* sehr *schwierig werden, weil man sich dann auch über die Wahrscheinlichkeitsverteilung der Eingaben (≙ „wie hoch ist die Wahrscheinlichkeit für eine bestimmte Eingabe“) Gedanken machen muss. Aus diesem Grund beschränken wir uns auf worst-case Analysen der vorgestellten Algorithmen.*

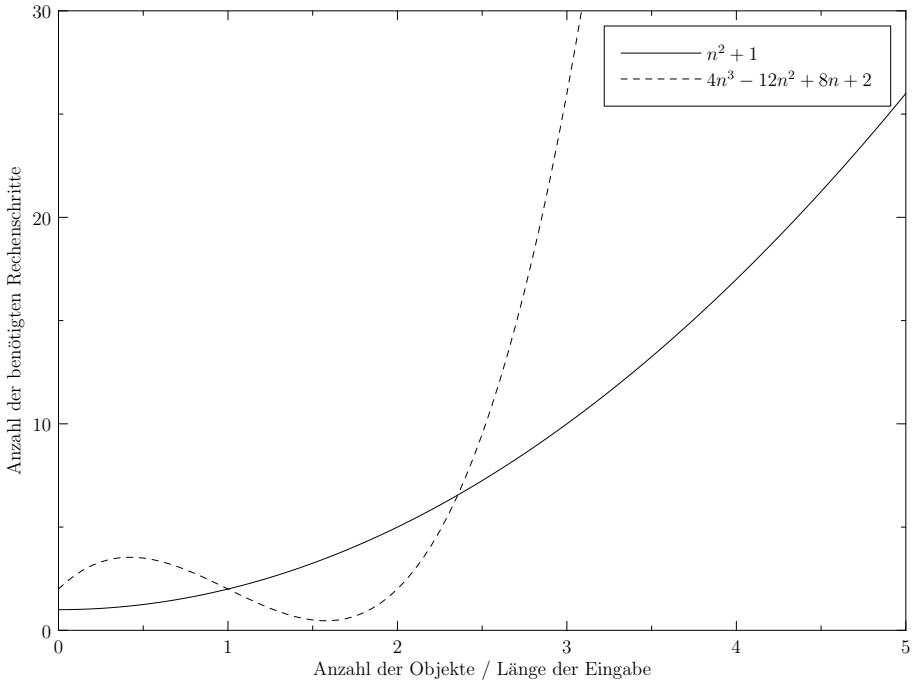


Abbildung 2: Beispiel für eine quadratische und eine kubische Funktion

2.2.2. Die O-Notation

Wir haben gesehen, dass die Laufzeit des Insertion Sorts $\frac{3}{2}n^2 + \frac{7}{2}n - 3$ beträgt. Um die Laufzeit von Algorithmen leichter vergleichen[4] und untersuchen zu können, benötigen wir eine gröbere Notation, um die doch relativ komplizierte Analyse zu vereinfachen.

Sei A ein Algorithmus zur Lösung eines Problems, dessen Laufzeit durch $f\colon \mathbb{N} \to \mathbb{N}$ beschränkt ist (worst-case). Weiterhin haben wir einen Algorithmus B zur Lösung eines (möglicherweise anderen) Problems gegeben, der die Laufzeit $g\colon \mathbb{N} \to \mathbb{N}$ benötigt. Wir wollen nun eine Beziehung einführen, die aussagt, ob A zur Lösung seines Problems im „Großen und Ganzen“ weniger oder höchstens genauso viel Zeit benötigt wie B. Die folgende Definition würde dies leisten[5]:

$$\forall n \in \mathbb{N}\colon f(n) \leq g(n)$$

Allerdings zeigt sich, dass diese Definition für unsere Zwecke noch nicht grob genug ist. Seien z.B. $f(n) = n^2+1$ und $g(n) = 4n^3-12n^2+8n+2$. Den Verlauf dieser Funktionen zeigt Abbildung 2, die auch veranschaulicht, dass für $n \geq 3$ immer $f(n) \leq g(n)$ gilt. Es ergibt sich hier aber ein Widerspruch zu unserer ursprünglichen Definition von „f ist im Großen und Ganzen nicht größer als g“, denn zwischen $n = 1$ und $n = 3$ ist der Funktionswert von g manchmal kleiner als der von f. Dieses Problem kann relativ leicht gelöst werden, da wir in der Informatik meist nicht an der Laufzeit eines Algorithmus für kleine Eingabelängen n interessiert sind. Wir legen nun fest: Eine Funktion f ist „für Informatiker im Großen und Ganzen[6] nicht größer als“ g, wenn ab einem bestimmten n_0 gilt: wenn $n \geq n_0$, dann ist $f(n) \leq g(n)$. Abbildung 2 zeigt für unseren Spezialfall, dass „f ist für Informatiker im Großen und Ganzen nicht größer als g“, da ab $n = 3$ der Funktionsgraph von g über dem von f verläuft.

[4] z.B. um zu entscheiden, welcher von zwei gegebenen Algorithmen eingesetzt werden soll

[5] Das Symbol $\forall n$ bedeutet dabei: „für alle n“. Genaueres entnehmen Sie Ihrer Formelsammlung bzw. Ihrem Mathematikskript.

[6] Man könnte auch sagen, dass die Funktion g fast immer („almost everywhere“) größer als f ist.

Unsere neue Idee, dass eine Funktion „für Informatiker im Großen und Ganzen nicht größer als“ eine andere Funktion ist, lässt sich durch eine leichte Veränderung der Definition wie folgt beschreiben: Eine Funktion f ist „im Großen und Ganzen für Informatiker nicht größer als“ g, wenn[7]

$$\exists n_0 \forall n \in \mathbb{N} \text{ gilt: falls } n \geq n_0, \text{ dann} f(n) \leq g(n)$$

Bei der praktischen Benutzung unserer neuen Definition tritt allerdings noch ein Nachteil auf, wie durch das folgende Beispiel verdeutlicht werden kann:

Beispiel 14: *Wir wollen einen Algorithmus analysieren und bewerten, der einen Skalar α mit einer $n \times n$ Matrix M multipliziert:*

$$\alpha \cdot M = \alpha \cdot \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} \alpha \cdot a_{11} & \dots & \alpha \cdot a_{1n} \\ \vdots & & \vdots \\ \alpha \cdot a_{n1} & \dots & \alpha \cdot a_{nn} \end{pmatrix}$$

Wir verwenden dazu Algorithmus 5 (wobei die quadratische Matrix M als zweidimensionales Array implementiert wird):

Eingabe: Quadratische Matrix M, Größe der Matrix n und Skalar α
Ergebnis: αM

```
for (i = 1 to n) do
    for (j = 1 to n) do
        M[i][j] = α · M[i][j];
    end
end
```

Algorithmus 5: Skalarmultiplikation einer quadratischen Matrix

Die Zeile „$M[i][j] = \alpha \cdot M[i][j]$“ wird offensichtlich n^2 mal ausgeführt. Nehmen wir an, dass ein Multiplikationsbefehl auf Prozessor P_1 genau 20 Takte braucht und auf Prozessor P_2 nur 2 Takte. Dann wäre die Laufzeit unseres Algorithmus auf Prozessor P_1 etwa $20n^2$ und auf Prozessor P_2 etwa $2n^2$, d.h. die Entscheidung, ob der Algorithmus „gut“ oder „geeignet“ ist, würde von der Hardware beeinflusst werden. Das macht aber so keinen Sinn, denn die Güte eines (abstrakten) Algorithmus und damit unsere Analyse und Beurteilung sollte sicherlich nicht von der realen Maschine abhängen. Neben der Hardwareabhängigkeit spielen auch andere Effekte eine Rolle. So ist die konkrete Rechenzeit u.a. von Details bei der Programmierung und von der Güte der Optimierungen durch den Compiler abhängig.

Beispiel 14 zeigt, dass wir einen Mechanismus benötigen, mit dem wir noch mehr „Details“ unterdrücken können, um unsere Analysen weiter zu vereinfachen. Offensichtlich benötigt die Anweisung $M[i][j] = \alpha \cdot M[i][j]$ eine konstante Zeit von c Takten, die uns aber nicht genau bekannt ist (und auch nicht bekannt sein braucht). Deshalb müssen wir unsere Notation „im Großen und Ganzen für Informatiker nicht größer als“ weiter anpassen, um von solchen konstanten Faktoren unabhängig zu werden. Diese veränderte Notation von „im Großen und Ganzen für Informatiker nicht größer als“ ist bekannt als O-Notation (nach Paul Bachmann, 1894) und wird in der Informatik sehr oft verwendet.

Definition 15: *Seien f, g Funktionen mit $f, g \colon \mathbb{N} \to \mathbb{N}$, dann ist*

$$O(g) =_{\text{def}} \{f \mid \exists c \, \exists n_0 \, \forall n \geq n_0 \text{ gilt } f(n) \leq c \cdot g(n)\}.$$

[7] Das Symbol $\exists n_0$ bedeutet dabei: „es gibt ein n_0 “. Genaueres entnehmen Sie Ihrer Formelsammlung bzw. Ihrem Mathematikskript. Eine kurze Einführung finden Sie in Abschnitt B.

Anschaulich bedeutet $f = O(g)$, dass f im Wesentlichen nicht schneller (stärker) wächst als g. Es gibt also eine Zahl n_0, ab der $c \cdot g(n)$ immer größer als $f(n)$ ist, wobei der Faktor c die oben beschriebenen „Hardwaredetails“ unterdrückt. Dies wird durch die Abbildungen 3 und 7 (siehe Seite 20) veranschaulicht.

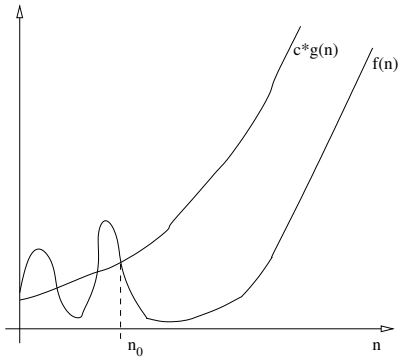


Abbildung 3: Die O-Notation anschaulich

Man bezeichnet $O(g)$ auch als asymptotische obere Schranke von f. Da $O(g)$ eine *Menge von Funktionen* darstellt, müsste man eigentlich $f \in O(g)$ schreiben. Dies erachten manche Benutzer der O-Notation als für zu umständlich, weshalb sich die Notation $f = O(g)$ durchgesetzt hat, obwohl sie unpräzise ist und in einigen Fällen mit großer Vorsicht benutzt werden muss (siehe Bemerkung 18). Will man diese Probleme vermeiden, so kann man einfach auf die Notation $f = O(g)$ verzichten und stattdessen immer $f \in O(g)$ verwenden.

Beispiel 16: *Es gilt* $\frac{3}{2}n^2 + \frac{7}{2}n - 3 = O(n^2)$*, da*

$$\underbrace{\frac{3}{2}n^2 + \frac{7}{2}n - 3}_{\triangleq f(n)} < \frac{7}{2}n^2 + \frac{7}{2}n + \frac{7}{2} \underbrace{\leq}_{\text{für } n > 0} \frac{7}{2}n^2 + \frac{7}{2}n^2 + \frac{7}{2}n^2 = \frac{21}{2}n^2$$

Damit haben wir $c = \frac{21}{2}$*,* $n_0 = 1$ *und* $\frac{3}{2}n^2 + \frac{7}{2}n - 3 = O(n^2)$*, d.h. Insertion Sort hat eine asymptotische Laufzeit von* $O(n^2)$[8]*.*

Beispiel 17: *Sei* $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_m n^m$ *und* $m \geq 1$*, dann gilt* $f \in O(n^m)$*, denn*

$$\begin{aligned} a_0 + a_1 n + \cdots + a_m n^m &\leq |a_0| + |a_1| n + \cdots + |a_m| n^m \\ &= (\tfrac{|a_0|}{n^m} + \tfrac{|a_1|}{n^{m-1}} + \cdots + \tfrac{|a_m|}{1}) n^m \underbrace{\leq}_{\text{für } n \geq 1} \overbrace{|a_0| + \cdots + |a_m|}^{c} = c \cdot n^m \end{aligned}$$

D.h. mit $c = \sum_{i=0}^{m} |a_i|$ *und* $n_0 = 1$ *gilt* $f(n) \in O(n^m)$*.*

Bemerkung 18: *Bei der Benutzung von „=“ in Bezug auf die O-Notation wird* nie $O(g(n)) = f(n)$ *geschrieben! Sonst würde ja z.B.* $2n^2 + n = O(n^2) = 3n^2 + 7$ *gelten, d.h.* $2n^2 + n = 3n^2 + 7$*, was offensichtlich absurd ist.*

Mit Hilfe der Definition können mit relativ wenig Aufwand (Übung!) die folgenden Rechenregeln gezeigt werden (z.B. [Knu97]):

1. $f(n) = O(f(n))$
2. $c \cdot O(f(n)) = O(f(n))$
3. $O(f(n)) + O(f(n)) = O(f(n))$
4. $O(O(f(n))) = O(f(n))$
5. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
6. $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$

[8] D.h. $\frac{3}{2}n^2 + \frac{7}{2}n - 3$ ist „im Großen und Ganzen nicht größer als“ n^2.

Bemerkung 19:

- *Die O-Notation „unterdrückt“ für große n unwichtige Details: Sei z.B. $\frac{n^2+\frac{3}{2}n-1}{n^2}$ gegeben. Für $n = 10$ ergibt sich der Wert 1.14 und für $n = 1\,000\,000$ ergibt sich 1.0000014999990, d.h. der Unterschied zwischen $n^2 + \frac{3}{2}n - 1$ und n^2 ist für große n gering (siehe Abbildung 4 auf Seite 18). Dies zeigt, dass die Definition der O-Notation gerechtfertigt ist, weil sich $n^2 + \frac{3}{2}n - 1 \in O(n^2)$ und $n^2 \in O(n^2)$ ergibt, d.h die O-Notation betrachtet $n^2 + \frac{3}{2}n - 1$ und n^2 als „gleichwertig“.*

- *Bei der Analyse des Insertion Sorts haben wir festgelegt, dass jeder Programmbefehl eine Zeiteinheit dauert. In der Praxis ist das unrealistisch (z.B. dauern Arrayzugriffe oft länger als Zuweisungen an eine Variable). Dadurch wird die Analyse noch komplizierter.*
  Wir legen fest: Eine elementare Anweisung benötigt eine konstante Zeit*, die nicht genauer spezifiziert wird (z.B. weil sie hardwareabhängig ist). Die O-Notation lässt alle diese Konstanten verschwinden und wir werden von Hardware, Implementations- und Compilerdetails unabhängig.*

- ***Achtung:** Selbst wenn g wesentlich schneller wächst als f, kann ein Algorithmus der Laufzeit $O(g)$ für bestimmte Eingaben schneller sein, als ein Algorithmus mit Laufzeit $O(f)$. So ist in Abbildung 3 in manchen Bereichen (wenn $n \leq n_0$) $g(n)$ kleiner als $f(n)$, wenn $c = 1$. Auch die Konstante c hat evtl. einen (sehr) großen Einfluss für (relativ) kleine n.*
  Für die Praxis bedeutet dies, dass man genau überlegen muss wie groß die Eingaben werden, um dann die Algorithmen genau vergleichen und bewerten zu können. Die Laufzeit von Programmen kann oft um konstante *Faktoren verbessert werden, wenn man Optimierungen in der Implementation vornimmt, bessere Compiler einsetzt oder effizientere Programmiersprachen verwendet (vgl. C vs. JAVA).*

- *Denken nicht vergessen!* $200ms$ *oder* $20sec$ *(konstante Faktoren können evtl. doch eine Rolle spielen) können in der Praxis wichtig sein! D.h. man sollte die Grenzen der O-Notation kennen.*

- *Ein Algorithmus der Laufzeit $O(p)$, wobei p ein beliebiges Polynom ist, heißt* Polynomialzeitalgorithmus.

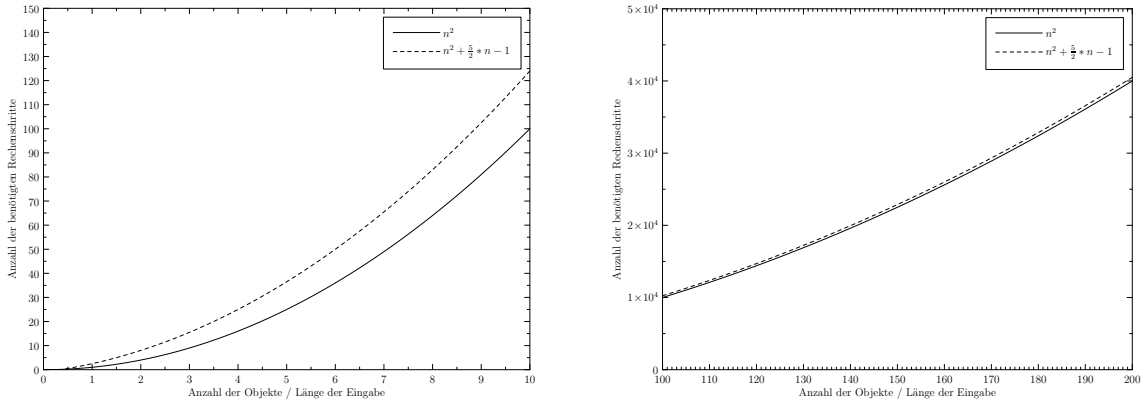
- *Es gibt (auch in der Praxis) Probleme für die keine Polynomialzeitalgorithmen bekannt sind oder existieren. (Stichwort: $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, wobei $\mathbf{P} \triangleq$ Polynomialzeit und $\mathbf{NP} \triangleq$ nichtdeterministische Polynomialzeit (siehe Abschnitt 6).*

- *Wir sagen: Ein Algorithmus mit Laufzeit*

  - $O(1)$ *hat „konstante Laufzeit“,*
  - $O(n)$ *hat „lineare Laufzeit“,*
  - $O(n \log n)$ *hat „quasilineare Laufzeit“,*
  - $O(n^2)$ *hat „quadratische Laufzeit“,*
  - $O(n^3)$ *hat „kubische Laufzeit“ und*
  - $O(2^n)$ *hat „exponentielle Laufzeit“.*

- *Ein „Gefühl“ für die Bedeutung von Laufzeiten vermittelt Abbildung 5.*

- *Wir haben die Laufzeit von Sortierverfahren in Abhängigkeit von der Anzahl der zu sortierenden Objekte angegeben. Bei der Analyse von Algorithmen ist es üblich, die Laufzeit in Abhängigkeit von der Länge der Eingabe (= Anzahl der Bits die benötigt werden um die Eingabe aufzuschreiben), die so genannte* Eingabelänge*, anzugeben. Solange wir Objekte fester Größe (z.B. `int`-Zahlen mit 32 Bit) sortieren, stimmt unser Vorgehen mit der*



(a) n^2 und $n^2 + \frac{5}{2}n - 1$ für „kleine“ n (b) n^2 und $n^2 + \frac{5}{2}n - 1$ für „große“ n

Abbildung 4: Vergleich des Wachstums der Funktionen n^2 und $n^2 + \frac{5}{2}n - 1$.

üblichen Praxis überein, denn die Anzahl der zu sortierenden Objekte unterscheidet sich dann nur um einen konstanten Faktor von der Eingabelänge. Diesen konstanten Faktor wird die O-Notation wieder „verdecken“. Die Eingabelänge oder eine Größe, die sich nur um einen konstanten Faktor von der Eingabelänge unterscheidet, wollen wir immer mit n bezeichnen.

- *Oft treten in (Sortier-)Algorithmen geschachtelte Schleifen auf, wobei jede Schleife $O(n)$-mal durchlaufen wird. Besteht der innerste Scheifenkörper nur aus konstant vielen elementaren Anweisungen, dann ergibt sich für eine k-fache Verschachtelung die Laufzeit*

$$\underbrace{O(n) \cdot O(n) \cdot \ldots \cdot O(n)}_{k-mal} \cdot \overbrace{c}^{Konstante} = O(n^k).$$

Als Anwendung dieser Überlegung ergibt sich in Beispiel 14 die Laufzeit $O(n^2)$.

- *Eine graphische Darstellung von einigen typischen Funktionen findet sich Abbildung 7 auf Seite 20.*

Mit den in Abbildung 6 angegebenen Regeln kann man die O-Notation auch dazu benutzen, die Laufzeit von komplexeren Algorithmen abzuschätzen. Dies soll durch das nächste Beispiel verdeutlicht werden:

Beispiel 20: *Gegeben sei ein zweidimensionales Feld. Jede Komponente dieses zweidimensionalen Feldes ist wiederum ein Feld, das unsortierte Zahlen enthält. Alle diese Felder sollen nun mit Insertion Sort sortiert werden. Dazu kommt der folgende Algorithmus zum Einsatz:*

Anzahl der Instruktionen	Objekte n					
	10	20	30	40	50	60
n	0.00001 Sekunden	0.00002 Sekunden	0.00003 Sekunden	0.00004 Sekunden	0.00005 Sekunden	0.00006 Sekunden
n^2	0.0001 Sekunden	0.0004 Sekunden	0.0009 Sekunden	0.0016 Sekunden	0.0025 Sekunden	0.0036 Sekunden
n^3	0.001 Sekunden	0.008 Sekunden	0.027 Sekunden	0.064 Sekunden	0.125 Sekunden	0.216 Sekunden
n^5	0.1 Sekunden	3,2 Sekunden	24.3 Sekunden	1.7 Minuten	5.2 Minuten	13.0 Minuten
2^n	0.001 Sekunden	1 Sekunde	17.9 Minuten	12.7 Tage	35.7 Jahre	366 Jahrhunderte
3^n	0.059 Sekunden	58 Minuten	6.5 Jahre	3855 Jahrhunderte	$2 \cdot 10^8$ Jahrhunderte	$1.3 \cdot 10^{13}$ Jahrhunderte

Abbildung 5: Rechenzeitbedarf von Algorithmen auf einem „1-MIPS“-Rechner

Eingabe: *Zweidimensionales Feld A[][] mit $n \times n$ Komponenten. Jede Komponente ist ein Feld mit n Zahlen.*
Ergebnis: *Zweidimensionales Feld A[][] mit $n \times n$ Komponenten. Jede Komponente ist ein* sortiertes *Feld mit n Zahlen.*

```
begin
    for (i = 0 to n − 1) do
        for (j = 0 to n − 1) do
            InsertionSort(Feld[i][j],n);
        end
    end
end
```

Algorithmus 6: Insertion Sort und zweidimensionale Felder

Kurzes Überlegen ergibt, dass die äußere Schleife n-mal durchlaufen wird. Da wir nur an einer groben Abschätzung interessiert sind (O-Notation!), können wir auch schreiben, dass die äußere Schleife $O(n)$-mal und damit der Rumpf der inneren Schleife $O(n^2)$-mal durchlaufen wird. Wir wissen schon, dass der Aufruf von Insertion Sort $O(n^2)$ viele Schritte benötigt. Mit den Rechenregeln für die O-Notation und den Anweisungen aus Algorithmus 6 ergibt sich eine Gesamtlaufzeit von $O(n \cdot n \cdot n^2) = O(n^4)$.

Angenommen, wir hätten einen neuen fiktiven Sortieralgorithmus `SuperSort` erfunden, der nur eine Laufzeit von $O(n \cdot \log n)$ benötigt. Tauschen wir Insertion Sort gegen unser neues Verfahren aus, so ergibt sich eine Laufzeit von $O(n \cdot n \cdot n \cdot \log n) = O(n^3 \cdot \log n)$.

Modifizieren wir unseren Algorithmus (aus hier nicht näher beschriebenen Gründen) so, dass die `for`-Schleifen nicht bis $n - 1$ laufen, sondern nur bis $(n - 1)/2$, dann ist die Anzahl der Durchläufe der äußeren Schleife immer noch $O(n)$ (konstante Faktoren werden unterdrückt!). Damit beträgt die Gesamtlaufzeit immer noch $O(n \cdot n \cdot n \cdot \log n) = O(n^3 \cdot \log n)$.

Algorithmenbaustein	Teilalgorithmen	Gesamtlaufzeit
Sequenz	Anweisung1; /* Laufzeit: f */ Anweisung2; /* Laufzeit: g */	$O(f) + O(g) = O(\max(f, g))$

Verzweigung	if (B) { /* Laufzeit für B: h */
	JAnweisung; /* Laufzeit: f */
	} else {
	NAnweisung; /* Laufzeit: g */
	}
		$O(h) + O(f) + O(g)$
		$=O(\max(h, f, g))$

Nichtabweisende Schleife	/* Max. Anzahl Durchläufe: m */
	/* Laufzeit für B: $O(f)$ */
	while (B) {
	Anweisung; /* Laufzeit: f */
	}
		$m \cdot O(f) = O(m \cdot f)$

Zählschleife	for ($i = 0$; $i < m$; i++) {
	Anweisung; /* Laufzeit: f */
	}
		$m \cdot O(f) = O(m \cdot f)$

Abbildung 6: Anwendung der O-Notation

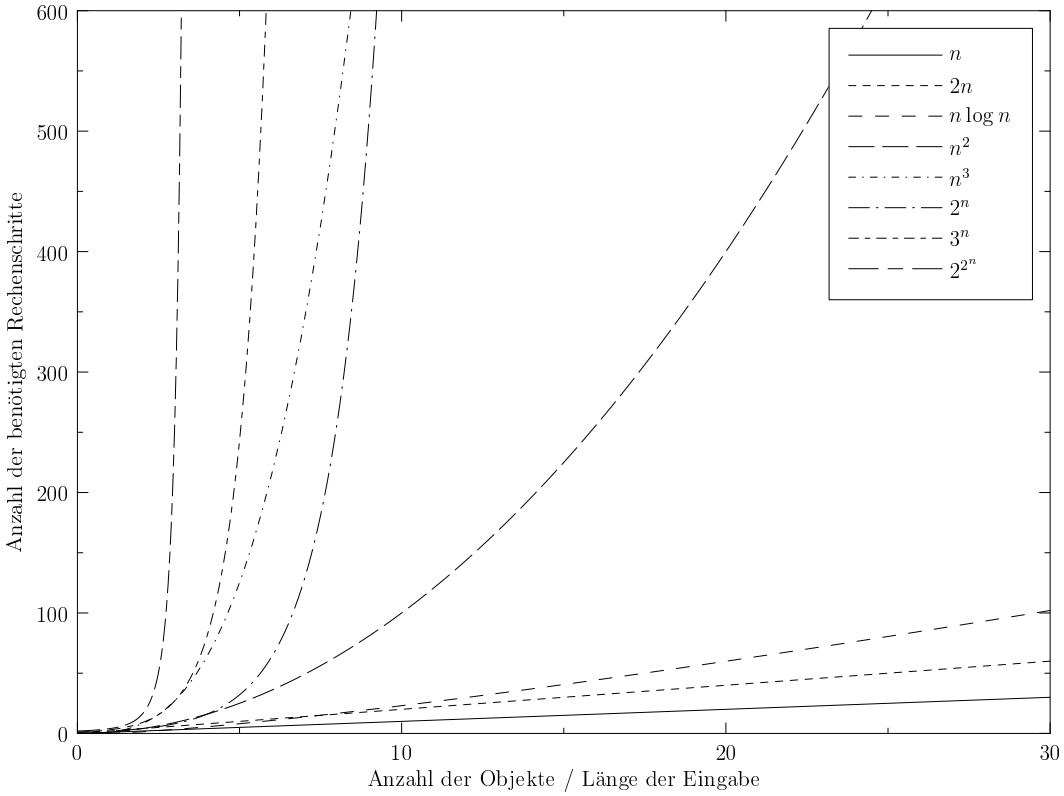


Abbildung 7: Das Wachstum von verschiedenen Funktionen

2.2.3. Selection Sort

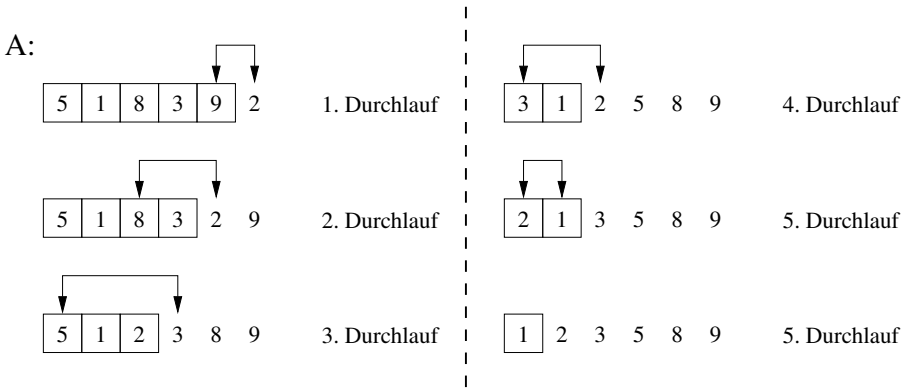
Die zentrale Idee des Selection Sorts ist es, aus der Liste der unsortierten Elemente das größte Element heraus zu picken und ganz an das Ende zu verschieben. Dannach wird der zu sortierende Bereich um ein Element verkleinert und die Auswahl bzw. Vertauschung eines größten Elements der restlichen unsortierten Elemente solange wiederholt, bis nichts mehr zu sortieren ist. Die Pseudocodedarstellung des Selection Sorts findet sich in Algorithmus 7.

Eingabe: Feld A mit n Komponenten
Ergebnis: Sortiertes Feld A' mit $A'[0] \leq A'[1] \leq A'[2] \leq \cdots \leq A'[n-1]$

```
p = n - 1;
while (p > 0) do
    g =„Index des größten Elements aus A im Bereich 0 ... p“;
    tausche A[p] und A[g];
    p = p - 1;
end
return A;
```

Algorithmus 7: Selection Sort

Beispiel 21: *Gegeben sei die Zahlenfolge* $5, 1, 8, 3, 9, 2$*, die mit Hilfe des Selection Sorts sortiert werden soll:*



Anhand der Graphik dieses Beispiels lässt sich genau erkennen, dass der unsortierte Bereich Schritt für Schritt verkleinert wird, wogegen der sortierte Bereich in jedem Schritt wächst.

Als Laufzeit für den Selection Sort ergibt sich: $O((n-1)+(n-2)+\cdots+1) = O(\sum_{i=1}^{n-1} i) = O(\frac{n(n-1)}{2}) = O(n^2)$ („quadratische Laufzeit“), da beim Suchen nach g erst $n-1$ Vergleiche anfallen und in den weiteren Schleifendurchläufen $n-2, n-3, \ldots, 1$ Vergleiche.

2.2.4. Bubble Sort

Hier ist die Idee das Array immer wieder zu durchlaufen und benachbarte Elemente zu vertauschen, die noch nicht der Sortierreihenfolge entsprechen.
Vorstellung: Schreibe das Array vertikal auf ein Blatt Papier, dann steigen „Blasen“ auf.

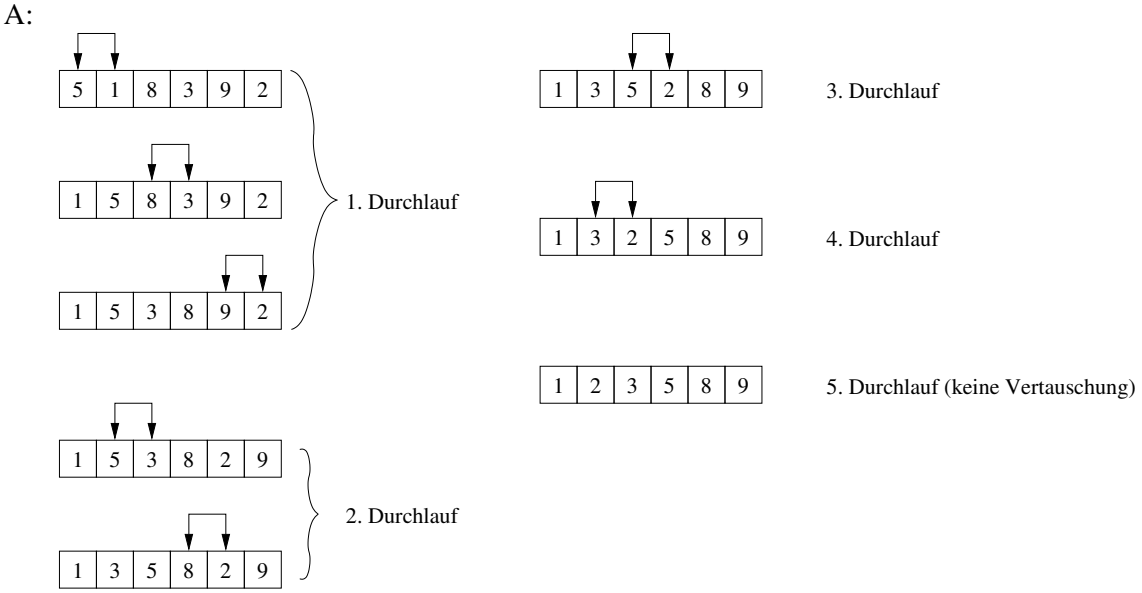
Pseudocodedarstellung des Bubble Sorts:

Eingabe: Feld A mit n Komponenten
Ergebnis: Sortiertes Feld A' mit $A'[0] \leq A'[1] \leq A'[2] \leq \cdots \leq A'[n-1]$

```
repeat
    for (i = 0 to n − 2) do
        if (A[i] > A[i + 1]) then
            tausche A[i] und A[i + 1];
        end
    end
until (es hat eine Vertauschung stattgefunden);
return A;
```

Algorithmus 8: Bubble Sort

Beispiel 22: *Gegeben sei die Zahlenfolge* $5, 1, 8, 3, 9, 2$ *die mit Hilfe des Bubble Sorts sortiert werden soll:*



Bemerkung 23: *Obwohl Bubble Sort mit einer Laufzeit von* $O(n^2)$ *auch nicht schneller ist als die bisher vorgestellten Verfahren, so hat er doch einen Vorteil: Die vertauschten Objekte liegen im Speicher direkt nebeneinander, d.h. die zu vertauschenden Daten liegen mit hoher Wahrscheinlichkeit schon im CPU-Cache, was eine schnelle Vertauschung ermöglicht. Sollten die Daten schon vorsortiert sein, dann kann Bubble Sort evtl. mit sehr wenigen Vertauschungen sehr schnell zum Ziel kommen (gut für vorsortierte Daten).*

2.3. Effiziente Sortierverfahren

Bisher haben wir nur Sortierverfahren kennengelernt, die eine Laufzeit von $O(n^2)$ haben. Ziel dieses Abschnitts ist es, Sortierverfahren kennen zu lernen, die mit einer Laufzeit von $O(n \log n)$ auskommen. Interessanterweise ist dies eine untere Schranke für Sortierverfahren, die Daten mit Hilfe eines Vergleichs von *zwei* Elementen bewerkstelligen (siehe Abschnitt 2.4).

2.3.1. Heap Sort

Die zentrale Datenstruktur des Heap Sorts ist der so genannte *Heap*[9]. Das Wort „Heap“ wird in der Informatik mit zwei völlig unterschiedlichen Bedeutungen benutzt:

- Heap als Datenstruktur, die von verschiedenen Algorithmen (z.B. Heap Sort, Prioritätswarteschlangen) verwendet wird.
- Heap als Speicherbereich in dem dynamische Objekte (vgl. `new`/`delete` in C++/JAVA bzw. `malloc`/`free` in C, vgl. Abschnitt 3.1.1 auf Seite 33) angelegt werden.

In diesem Abschnitt wird Heap nur als Begriff für die Datenstruktur verwendet.

Definition 24 („Heapbedingung“): *Eine Folge $F = a_l a_{l+1} a_{l+2} \dots a_r$ heißt* Heap*, wenn die beiden folgenden Bedingungen eingehalten werden:*

Für alle i mit $l \leq i \leq r$ gilt,

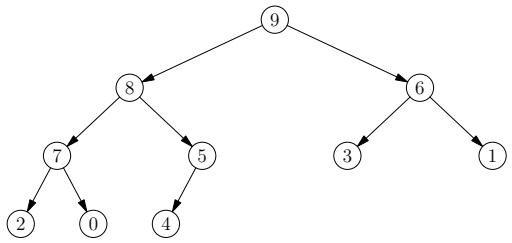
i) wenn $2i \leq r$, dann ist $a_i \geq a_{2i}$ und

ii) wenn $2i + 1 \leq r$, dann ist $a_i \geq a_{2i+1}$.

Das Element a_{2i} (bzw. a_{2i+1}) heißt Nachfolger *(oder* Kind*) von a_i, und a_i wird als* Vorgänger *(oder* Vater*) von a_{2i} und a_{2i+1} bezeichnet. Der Knoten ohne Vorgänger heißt* Wurzel *und alle Knoten ohne Nachfolger sind* Blätter *des Heaps (vgl. Abschnitt 3.2 auf Seite 45).*

Beispiel 25: *Die durch die folgende Tabelle gegebene Folge ist ein Heap:*

i	1	2	3	4	5	6	7	8	9	10
a_i	9	8	6	7	5	3	1	2	0	4



Jeder Heap kann anschaulicher auch graphisch dargestellt werden, wobei diese Darstellung eines Heaps als Binärbaum *bekannt ist. Aus diesem Grund kann man sich einen Heap auch als Binärbaum vorstellen, der in einem Array gespeichert wurde. Die Folge*

i	1	2	3	4	5	6	7	8	9	10
a_i	9	8	6	2	5	3	1	7	10	4

hingegen ist kein Heap, da $a_4 < a_8$ und $a_4 < a_9$ gilt.

Beobachtung 26: *Die Elemente $a_{\lfloor n/2 \rfloor + 1}, \dots, a_n$ einer Folge $a_1, \dots, a_n$ erfüllen automatisch die Heapbedingung, denn sie haben keine Nachfolger, d.h. sie sind Blätter. Aus diesem Grund bilden alle diese Knoten je einen Heap für sich, der nur aus einem Knoten besteht.*

Das nächste Ziel ist die Entwicklung einer Funktion `MaxHeapify`, die einen „defekten“ Heap repariert, wenn nur die Wurzel die Heapbedingung verletzt und alle anderen Knoten die Heapbedingung erfüllen.

Die zentrale Idee dabei ist es, die Wurzel mit dem größeren der beiden Nachfolger zu vertauschen, denn dann erfüllt der Wurzelknoten die Heapbedingung. Allerdings kann es nun sein, dass der neue Nachfolger der Wurzel, also die ursprüngliche Wurzel, die Heapbedingung verletzt. Aus

[9] Die Übersetzung von Heap lautet „Halde“ oder „Haufen“.

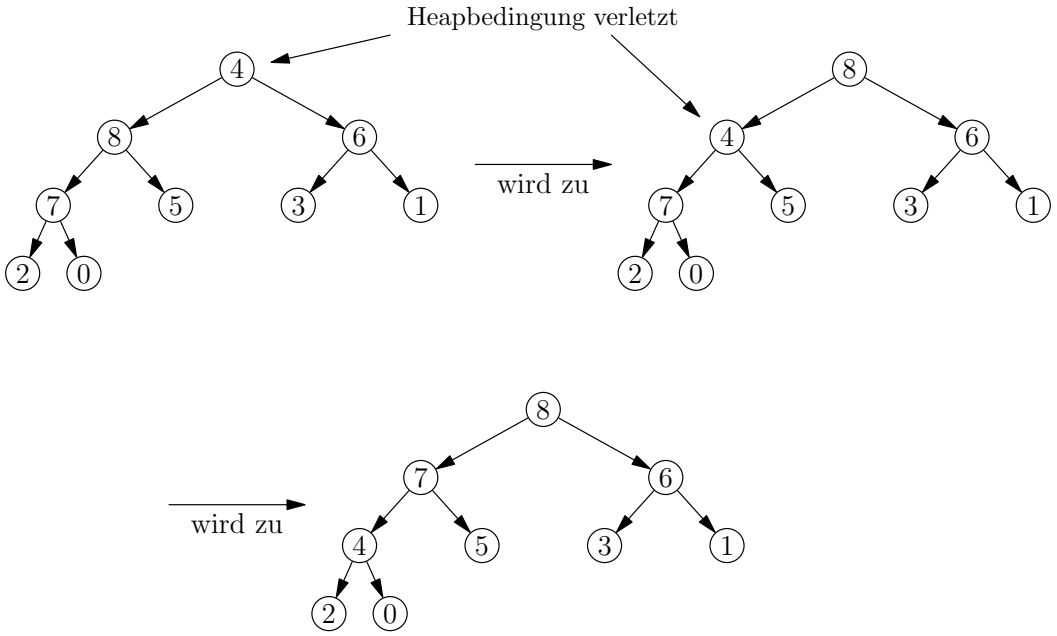


Abbildung 8: Schritte bei der Reparatur eines defekten Heaps

diesem Grund muss dieser Vertauschungsprozess solange durchgeführt werden, bis die Heapbedingung durch alle Knoten erfüllt ist, was spätestens dann erreicht ist, wenn die ursprüngliche Wurzel mit einem Blatt vertauscht wird. Anschaulich kann man sich vorstellen, dass die Wurzel im Baum „versickert“ bis der richtige Platz erreicht ist. Ein Beispiel für die notwendigen Schritte einen solchen defekten Heap zu reparieren zeigt Abbildung 8.

Dies führt zu Algorithmus 9, wenn die Folge $a_l, \ldots, a_r$ als Array A gespeichert wird und für das r bzw. l zulässige Indizes sind. Mit Hilfe der Funktion `MaxHeapify` kann nun eine Folge $a_1, \ldots, a_n$ schrittweise in einen Heap gewandelt werden. Dazu kann man beobachten, dass die Elemente $a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$ ja schon die Heapbedinung erfüllen, denn sie haben keine Nachfolger, die die Heapbedingung verletzen könnten. Nun werden die Elemente $a_{\lfloor n/2 \rfloor}, \ldots, a_1$ schrittweise mit der Hilfe von `MaxHeapify` in die Heaps aufgenommen (siehe Algorithmus 10), d.h. nach Abarbeitung von `ConstructHeap` erfüllen alle Elemente $a_1, \ldots, a_n$ die Heapbedingung und bilden damit einen einzigen Heap.

Es ist unmittelbar einsichtig, dass das größte in einem Heap gespeicherte Element die Wurzel des Heaps ist (vgl. mit dem letzten Baum in Abbildung 8). Mit dieser Beobachtung kann nun leicht ein Sortieralgorithmus formuliert werden, der als HeapSort (vgl. Algorithmus 11) bekannt ist. Dazu wird die Wurzel ($\triangleq$ größtes Element) mit dem letzten Element in der zu sortierenden Folge vertauscht. Danach wird der zu sortierende Bereich von rechts nach links um ein Element verkleinert und die Heapbedinung mit `MaxHeapify` repariert, da der neue Wurzelknoten evtl. die Heapbedinung verletzt.

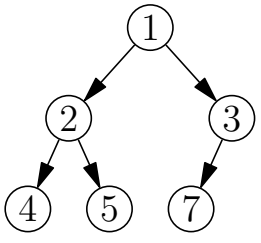


Abbildung 10: Ein Heap der Höhe 3

Beispiel 27: *Es soll die Eingabefolge* $3, 7, 9, 1, 5, 2, 4$ *mit sieben Elementen mit Hilfe von HeapSort sortiert werden. Die einzelnen Schritte zeigt Abbildung 9. Dabei wird in den Schritten* 1-4 *aus der Eingabefolge ein Heap erzeugt und in den Schritten* 5-10 *wird sortiert, wobei sich der*

Eingabe: Ein Array A mit dem zulässigen Indexbereich von l nach r
Ergebnis: Das Array A bei dem alle Knoten die Heapbedingung erfüllen (Heap)

```
                                                    /* Wurzel des Heaps ganz links */
i = l;
                                                          /* Wert der Wurzel holen */
x = A[l];
                                                /* Position des linken Nachfolgers */
j = 2 · l;
                        /* Wenn beide Nachfolger existieren, dann hole den größeren */
if ((j < r) && (A[j] < A[j + 1])) then
    |                                             /* Index des rechten Nachfolgers */
    j=j+1;
end
                                 /* Solange noch im Bereich und Heapbedingung verletzt */
while ((j ≤ r) && (x < A[j])) do
    |                                        /* Größeren Nachfolger nach oben kopieren */
    A[i] = A[j];
    |                                          /* Neue Wurzel ist der Nachfolger */
    i = j;
    |                            /* Neuen linken Nachfolger der aktuellen Wurzel wählen */
    j = 2 · j;
    |                    /* Wenn beide Nachfolger existieren, dann holen den größeren */
    if ((j < r) && (A[j] < A[j + 1])) then
        |                                         /* Index des rechten Nachfolgers */
        j=j+1;
    end
    |                                  /* Ursprüngliche Wurzel an den richtigen Platz */
    A[i] = x;
end
```

Algorithmus 9: `MaxHeapify`

Eingabe: Ein Array A bei dem nur der Knoten $A[l]$ die Heapbedingung verletzt
Ergebnis: Das Array A bei dem alle Knoten die Heapbedingung erfüllen (Heap)

```
                        /* Alle Knoten von ⌊n/2⌋ + 1 bis n erfüllen die Heapbedingung */
l = ⌊n/2⌋ + 1;
r = n;
while (l > 1) do
    |                                                  /* Neuen Knoten aufnehmen */
    l = l − 1;
    |                                        /* Neuen Knoten in den Heap einbauen */
    MaxHeapify(A, l, r);
end
```

Algorithmus 10: `ConstructHeap`

Eingabe: Das zu sortierende Array A
Ergebnis: Das sortierte Array A'

```
                    /* Wandle Array A in einen Heap um (Konstruktionsphase) */
ConstructHeap(A, 1, n);
l = 1;
r = n;
     /* Heap von rechts nach links verkleinern (Abbau- und Sortierphase) */
while (r > 1) do
    vertausche A[l] und A[r];
                                                        /* Heap verkleinern */
    r = r − 1;
                                                  /* Heap wieder verkleinern */
    MaxHeapify(A, l, r);
end
                                                 /* Sortiertes Array zurückgeben */
return A;
```

Algorithmus 11: `HeapSort`

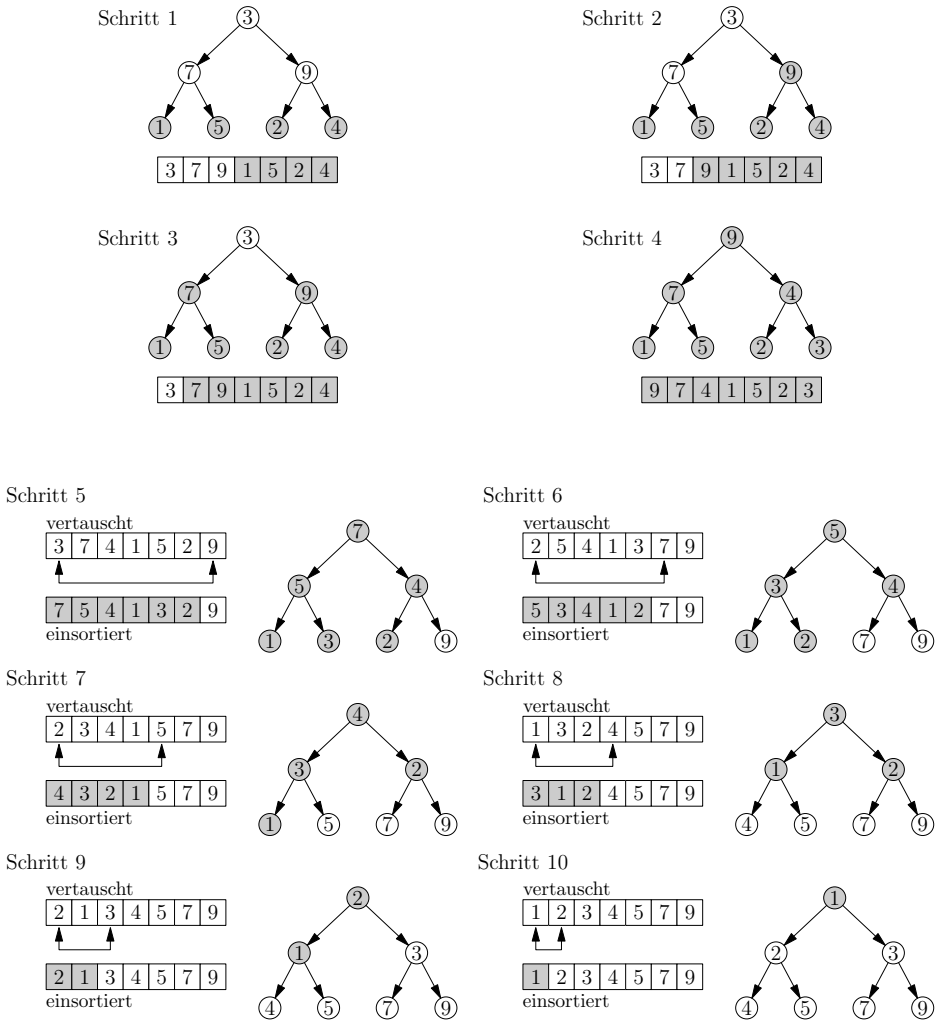


Abbildung 9: Schritte bei der Durchführung von HeapSort

Heap schrittweise immer weiter verkleinert. Die grau hinterlegten Knoten bzw. Arraykomponenten zeigen, welche Elemente zu diesem Zeitpunkt in einem Heap gespeichert sind.

In jeden Baum gibt es für einen beliebigen Knoten immer einen eindeutigen Pfad, der an der Wurzel startet und an diesem Knoten endet. Für die Untersuchung der Laufzeit des HeapSorts wird die folgende Definition benötigt:

Definition 28: *Sei $F = b_1 b_2 \dots b_m$ eine Folge von Knoten in einem Heap, wobei b_{i+1} der Nachfolger des Knotens b_i ist, und sei F eine* längste *mögliche Folge in diesem Heap. Die Zahl m heißt dann* Höhe *des Heaps.*

Durch Definition 28 ergibt sich für den Heap aus Abbildung 10 eine Höhe von 3, wie die Knotenfolge $1, 3, 7$ zeigt. Mit Hilfe der Definition der Höhe eines Heaps ergibt sich unmittelbar die folgende Aussage, die leicht mit Hilfe der vollständigen Induktion gezeigt werden kann:

Satz 29: *Ein Heap der Höhe h hat maximal $2^h - 1$ Knoten.*

Beweis: Wir führen eine Induktion über die Höhe h des Heaps:
(IA) Ein Heap der Höhe 1 enthält genau einen Knoten und es gilt $2^1 - 1 = 1$, d.h. der Induktionsanfang ist erfüllt
(IV) Ein Heap der Höhe h enthält maximal $2^h - 1$ Knoten.
(IS) $h \to h + 1$: Durch Benutzung der Induktionsvoraussetzung ergibt sich, dass ein Heap der Höhe $h + 1$ maximal

$$\underbrace{2^h - 1}_{\text{linker Teilheap}} + \underbrace{2^h - 1}_{\text{rechter Teilheap}} + \underbrace{1}_{\text{Wurzel}} = 2^{h+1} - 1$$

Knoten enthält. #
Damit gilt für die Höhe h und die Anzahl der Knoten n eines Heaps der Zusammenhang:

$$2^{h-1} \leq n \leq 2^h - 1$$

Dies führt dann zu der nächsten Folgerung:

Folgerung 30: *Ein Heap mit n Knoten hat die Höhe $O(\log n)$. Der Aufwand von* `MaxHeapify` *für eine Eingabefolge mit n Knoten beträgt $O(\log n)$.*

Damit ergibt sich für den Zeitbedarf des HeapSorts:

$$n/2\, O(\log n) + n\, O(\log n) = O(n \log n) + O(n \log n) = O(n \log n)$$

2.3.2. Merge Sort

Der Merge Sort ist ein schönes Beispiel für ein Sortierverfahren, das den „teile und herrsche“-Ansatz verwendet.

- **Divide:** Ein Array wird in zwei Teilarrays mit je $n/2$ Elementen aufgeteilt.
- **Conquer:** Sortiere die zwei Subarrays mit Merge Sort.
- **Combine:** Setze die zwei sortierten Felder wieder zusammen.

Wir benötigen also einen Algorithmus (vgl. Algorithmus 12), der zwei sortierte Zahlenfolgen zu einer sortierten Zahlenfolge zusammensetzt (merge). Mit Hilfe der merge-Operation können wir nun den rekursiven Merge Sort (vgl. Algorithmus 13) formulieren. Als Abbruchbedingung für die Rekursion verwenden wir, dass einelementige Folgen automatisch schon sortiert sind.

Eingabe: Zwei sortierte Folgen F_1 und F_2
Ergebnis: Eine sortierte Folge F, die die Elemente von F_1 und F_2 enthält

```
F = leere Folge;
                                   /* Solange in beiden Listen noch Elemente sind */
while ((F1 nicht leer) && (F2 nicht leer)) do
    |                  /* Prüfe, ob der Head von F1 kleiner ist als der von F2 */
    if (Anfangselement von F1 ist kleiner als Anfangselement von F2) then
    |                                                  /* Entferne den Kopf aus F1 */
    |   t = Anfangselement von F1;
    |   lösche Anfangselement von F1;
    else
    |                                                  /* Entferne den Kopf aus F2 */
    |   t = Anfangselement von F2;
    |   lösche Anfangselement von F2;
    end
    hänge t an F an;
end
hänge eine evtl. verbliebene Restfolge an F an;
                                                 /* Die Liste F ist nun sortiert */
return F;
```

Algorithmus 12: merge-Operation

Eingabe: Eine unsortierte Folge F
Ergebnis: Eine sortierte Folge F', die alle Elemente von F enthält

```
if ( F hat genau ein Element ) then
    |                        /* Eine Liste mit einem Element ist immer sortiert */
    return F;
else
    |          /* Hier wird die Liste im Divide-Schritt in zwei Teile geteilt */
    halbiere F in F1 und F2;
    |           /* Sortierte rekursive die zwei Teillisten (Herrsche-Schritt) */
    F1 = Merge Sort(F1);
    F2 = Merge Sort(F2);
    |              /* Verbinde sortierte Teillisten zu einer sortierten Liste */
    return merge(F1, F2);
end
```

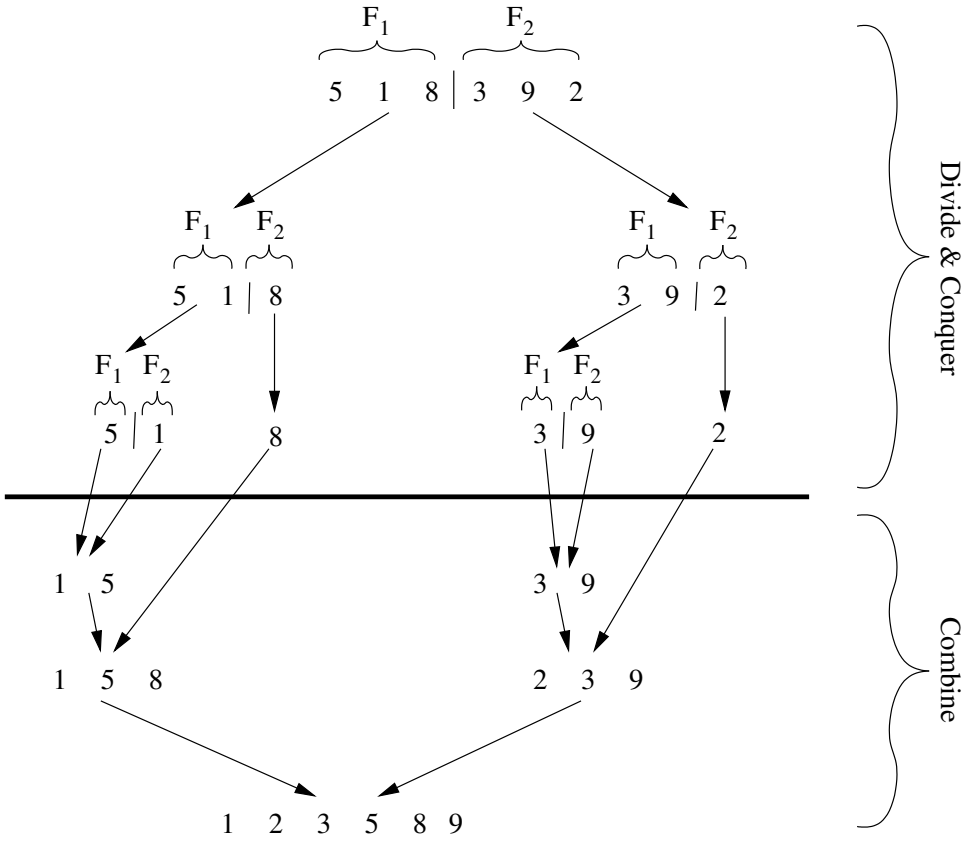
Algorithmus 13: Merge Sort

Bemerkung 31:

Die Funktion „merge“ benötigt bei einer einfachen Implementierung den doppelten Speicherplatz (für die merge-Operation), d.h. Merge Sort benötigt relativ viel Speicher.

Eine interessante Eigenschaft von Merge Sort ist, dass nie alle Daten im Hauptspeicher gehalten werden müssen. Stattdessen zerteilt man die Folgen rekursiv solange bis sie in den Hauptspeicher passen, sortiert sie und lagert wieder auf einen externen Speicher aus. Danach werden die (sortierten) Teilfolgen wieder (parallel) eingelesen, gemischt und wieder in den externen Speicher geschrieben. Sortierverfahren die mit externen Speichern arbeiten können, nennt man auch externe Sortierverfahren*, wogegen Sortierverfahren, die alle Daten im Hauptspeicher halten müssen,* interne Sortierverfahren *genannt werden.*

Beispiel 32: *Es soll die Folge* $5, 1, 8, 3, 9, 2$ *mit Hilfe von Merge Sort sortiert werden. Dabei ergibt sich das folgende Bild:*



Die zwei Phasen entstehen durch die rekursive Struktur des Algorithmus.

Die Laufzeit von Merge Sort Aufgrund der rekursiven Struktur von Merge Sort ist die Analyse nicht so einfach. Aus diesem Grund beschränken wir uns darauf die Anzahl der Vergleiche von zwei Elementen zu zählen.
Sei $V(n) =_{\text{def}}$ „Die Anzahl der Vergleiche beim Sortieren von n Elementen“, dann gilt:

$$\begin{aligned} V(n) &= \underbrace{2 \cdot V(n/2)}_{\text{sortieren von } F_1 \text{ und } F_2} + \overbrace{n}^{\text{„merge“}}, \text{ für } n \geq 2 \\ V(1) &= 0 \end{aligned}$$

Der Extremfall für n Vergleiche im merge-Schritt tritt auf, wenn eine Liste das größte Element und die andere alle anderen Elemente enthält. Unser Ziel ist, die Gleichung für V in eine explizite Form zu bringen. Dazu nehmen wir vereinfachend an, dass n eine 2er Potenz ist (damit wir immer in zwei gleichgroße Stücke aufteilen können).
Sei also $n = 2^N$, dann gilt:

$$\begin{aligned} & V(2^N) && = 2V(2^{N-1}) + 2^N \\ \Leftrightarrow \quad & \tfrac{V(2^N)}{2^N} && = \underbrace{\frac{V(2^{N-1})}{2^{N-1}}}_{(*)} + 1 \end{aligned}$$

Wir ersetzen nun Schritt für Schritt den Term $(*)$, also $\frac{V(2^{N-i})}{2^{N-i}}$ solange bis $i = N$ und erhalten

$$\frac{V(2^N)}{2^N} = \frac{V(2^{N-1})}{2^{N-1}} + 1 = \frac{V(2^{N-2})}{2^{N-2}} + 1 + 1 = \cdots = \frac{V(2^0)}{2^0} + \cdots + 1 = N$$

Da $n = 2^N$, gilt $N = \log_2 n$ und $V(2^N) = N \cdot 2^N$, also $V(n) = n \log_2 n$. Eine praktische Implementierung benutzt pro Vertauschungsoperation nur konstant viele andere Befehle, d.h. die Laufzeit von Merge Sort beträgt $O(n \log_2 n)$.

Bemerkung 33: *Eine Gleichung (bzw. Ungleichung), die eine Funktion mit Hilfe von Funktionswerten für kleinere Eingaben beschreibt, nennt man* Rekurrenzgleichung. *Anschaulich bedeutet dies, dass das Funktionssymbol auf der rechten und linken Seite der Gleichung (bzw Ungleichung) vorkommt.*

In der Literatur, z.B. [TCRC01], Kapitel 4, sind vielfältige Techniken zur Lösung solcher Gleichungen beschrieben. Dort findet sich auch das „Master Theorem“:

Satz 34: *Seien $a \geq 1$ und $b \geq 1$ Konstanten. Weiterhin sei $f(n)$ eine Funktion, dann gilt für die Funktion $T\colon \mathbb{N} \to \mathbb{N}$, die gegeben ist durch*

$$T(n) = a \cdot T(n/b) + f(n),$$

die folgende asymptotischen Beziehungen:

1. *Wenn $f(n) = O(n^{\log_b a - \epsilon})$ und die Konstante $\epsilon > 0$, dann $T(n) = O(n^{\log_b a})$.*
2. *Wenn $f(n) = O(n^{\log_b a})$, dann $T(n) = O(n^{\log_b a} \log_2 n)$.*

Wählen wir $a = 2$ und $b = 2$, dann ergibt sich mit $f(n) = n$ und der zweiten Aussage des Mastertheorems die Anzahl der Vergleiche von Merge Sort ganz automatisch.

2.3.3. Quick Sort

Auch dieses Sortierverfahren verwendet das „teile und herrsche“ Prinzip und eine Analyse ergibt, dass die Anzahl der Vertauschungen im *Durchschnitt* der von Merge Sort entspricht (also $O(n \log n)$. Allerdings entfällt beim Quick Sort das Hilfsfeld, das Merge Sort benötigt, da beim Quick Sort der Mischvorgang vermieden wird.
Idee:
Zerteile die Folge in zwei Hälften, wobei die linke Teilfolge alle Elemente enthält, die kleiner als ein Referenzelement sind. Die andere Teilfolge enthält alle Elemente, die größer als das Referenzelement sind. Dieses Referenzelement wird auch oft *Pivot-Element* (kurz: *Pivot*) bezeichnet. Typischerweise wird oft das mittlere Element der Folge verwendet, aber auch die erste oder die letzte Komponente kann benutzt werden.

Es ergibt sich die folgende Pseudocodedarstellung des Quick Sorts:

Eingabe: Eine unsortierte Folge F, untere Grenze *links* und eine obere Grenze *rechts*
Ergebnis: Eine sortierte Folge F', die alle Elemente von F enthält

```
l = links;
r = rechts;
if (r > l) then
    /* Mittleres Element als Pivot verwenden */
    pivot = F[(l + r) / 2];
    while (l ≤ r) do
        /* Suche Element das nicht in die linke Folge gehört */
        while (F[l] < pivot) do
            l++;
        end
        /* Suche Element das nicht in die rechte Folge gehört */
        while (F[r] > pivot) do
            r--;
        end
        /* Teste ob wir Tauschkandidaten gefunden haben */
        if (l ≤ r) then
            tausche F[l] und F[r];
            l++;
            r--;
        end
    end
end
/* Teste ob linke (unsortierte) Teilliste nicht leer */
if (links < r) then  QuickSort(F, links, r) };
;
/* Teste ob rechte (unsortierte) Teilliste nicht leer */
if (r < rechts) then  QuickSort(F, l, rechts) };
;
return F;
```

Algorithmus 14: Quick Sort

Bemerkung 35: *Der Quick Sort kann auch entarten, wenn aus „Zufall“ das Pivotelement immer so gewählt wird, dass eine der beiden Teilfolgen leer bleibt.*
Bsp.: Die Eingabe ist schon sortiert und wir wählen immer das erste Element als Pivot, dann ist F_1 immer leer und F_2 enthält alle anderen Elemente außer dem Pivot. D.h. wir benötigen so viele rekursive Aufrufe wie in der Eingabefolge Elemente vorhanden sind (also n Schritte), weil der divide-Schritt nicht in zwei etwa gleichgroße Teile aufspaltet. Eine Analyse ergibt, dass dann die Laufzeit $O(n^2)$ beträgt (es werden noch etwa n Schritte benötigt um die Listen aufzuteilen). D.h. es gibt Fälle in denen Bubble Sort oder Insertion Sort schneller sind als Quick Sort (Bubble Sort bricht schon nach n Schritten ab, wenn die Eingabe aus n Objekten besteht und sortiert ist).

Trotz dieses Nachteils ist Quick Sort in der Praxis ein sehr wichtiger Sortieralgorithmus (vgl. `qsort()` in der C-Standardlibrary).

2.4. Eine untere Schranke für vergleichsbasierte Sortierverfahren

Wir haben nun Sortierverfahren die n Objekte in der Zeit $O(n \log n)$ sortieren. Dies ist eine deutliche Verbesserung zu den Verfahren mit Laufzeit $O(n^2)$.

Nun stellt sich natürlich die Frage, ob man nicht einen noch schnelleren Algorithmus für das Sortierproblem finden kann, der z.B. in der Zeit $O(n)$ arbeitet. Unmittelbar klar ist, dass das Sortierproblem nicht in weniger als Laufzeit n gelöst werden kann, denn unterhalb dieser Schranke können wir gar nicht alle Elemente der Eingabe „ansehen“.

Wir wollen in diesem Abschnitt zeigen, dass kein Algorithmus existiert, der mit Vergleichen von zwei Elementen arbeitet und der schneller als $O(n \log n)$ ist. Eine solche Aussage über ein Berechnungsproblem nennt man *untere Schranke*[10] für das Problem. Aussage dieser Art sind selten in der Theorie der Informatik, da wir ja unendlich viele Algorithmen (die wir sicherlich nicht alle kennen und erst recht nicht alle aufschreiben können) in unsere Untersuchungen mit einbeziehen müssen. Obere Schranken[11] für die Komplexität eines Problem sind dagegen einfach zu gewinnen: Entwerfe einen Algorithmus für das gegebene Problem und analysiere die Laufzeit. Wir wissen z.B. das $O(n \log n)$ eine obere Schranke für das Problem SORT ist.

Wir nehmen an, dass unsere Sortierverfahren nur durch Fragen der Form $A[i] \stackrel{?}{<} A[j]$ etwas über die Eingabe erfahren. Das war bei allen Verfahren in diesem Kapitel so.

Weiterhin haben wir gesagt: Für das Problem SORT suchen wir eine Permutation π, die alle Eingabeelemente aufsteigend anordnet, d.h. es gilt $A[\pi(1)] < A[\pi(2)] < A[\pi(3)] < \cdots < A[\pi(n)]$. Dabei bedeutet die Permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ 3 & 5 & 1 & \cdots & 2 \end{pmatrix}$$

von n Objekten, dass das erste Element durch das dritte (also $\pi(1) = 3$), das zweite durch das fünfte (also $\pi(2) = 5$), das dritte durch das erste (also $\pi(3) = 1$) und das $n$te durch das zweite Element (also $\pi(n) = 2$) ersetzt wird.

Wieviele Permutationen gibt es? Da wir n Elemente vertauschen, muss jedes Element genau *einmal* in der unteren Zeile einer Permutation vorkommen, d.h. wir haben für die erste Position n Möglichkeiten, für die zweite $n-1$ und für die letzte Position genau eine Möglichkeit. Damit gibt es $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = n!$ viele verschiedene Permutationen von n Elementen.

Ein Sortieralgorithmus muss mit allen möglichen Eingaben umgehen (sortieren) können, d.h. er muss jede mögliche Permutation der Eingabedaten berücksichtigen können.

Der Ablauf eines beliebigen Sortierverfahren, dass Vergleiche der Art $A[i] \stackrel{?}{<} A[j]$ durchführt, kann dann als *Entscheidungsbaum* gezeichnet werden (siehe Abbildung 11).

In einem solchen Entscheidungsbaum stellt ein Pfad von der Wurzel zu einem Blatt eine Berechnung für eine konkrete Eingabe dar. So gibt der dick gezeichnete Pfad die Berechnung an, die zu der Permutation führt, die die Eingabe `A[] = {6, 8, 5}` sortiert, denn `A[3]` $= 5$, `A[1]` $= 6$, `A[2]` $= 8$ ist dann aufsteigend angeordnet.

Der längste Pfad von dem Wurzelknoten zu einem Blatt ($\triangleq$ Permutation) in einem Baum nennt man *Tiefe des Baumes*. Ein Entscheidungsbaum unseres Typs (zwei Verzweigungen pro Knoten) der Tiefe s hat maximal 2^s Blätter, denn in jeder „Ebene“ des Baums verdoppelt sich die Anzahl der Knoten höchstens. Damit erhalten wir

[10] D.h. jeder Algorithmus, der das Sortierproblem durch Vergleichen von zwei Elementen löst, braucht *mindestens* $O(n \log n)$ Schritte.

[11] Dies bedeutet, dass wir nicht mehr als $O(n \log n)$ Schritte zur Lösung des Sortierproblems brauchen, denn wir kennen ja schon Algorithmen, die dies leisten.

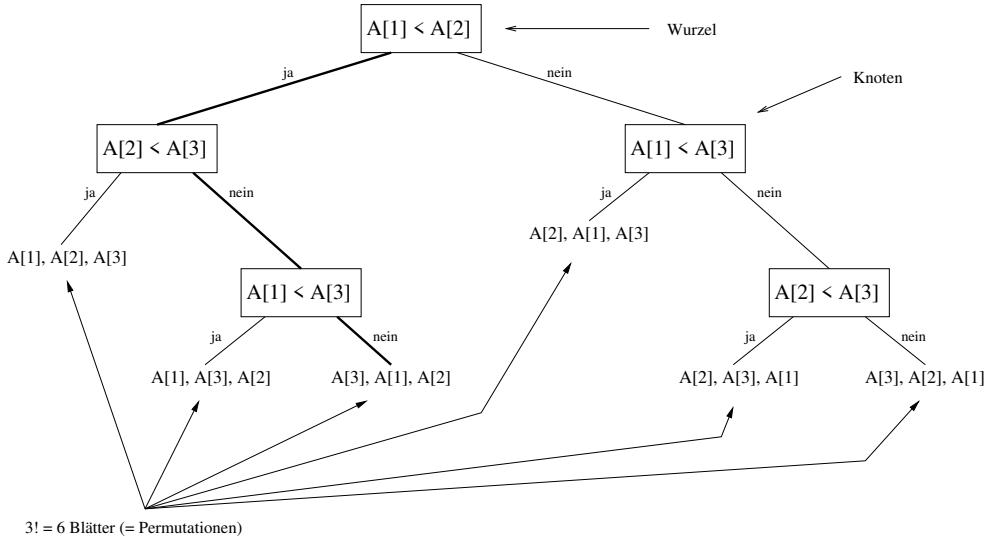


Abbildung 11: Entscheidungsbaum für den Fall $n = 3$ (Anordnung wie bei Insertion Sort)

$$
\begin{array}{rrll}
 & \overbrace{2^s}^{\text{Anzahl der Blätter}} & \geq & \overbrace{n!}^{\textit{Permutationen}} \\
\Leftrightarrow & s & \geq \log_2 n! & = \underbrace{\frac{1}{\ln 2}}_{=c} \cdot \ln n! \\
 & & \text{Stirlingsche Formel} \rightarrow & \approx c \cdot ((n+\tfrac{1}{2}) \ln n - n + \underbrace{\ln \sqrt{2\pi}}_{\text{konstant}} \\
 & & & = O(c' \cdot n \log_2 n) = O(n \log_2 n)
\end{array}
$$

D.h. ein Entscheidungsbaum, der alle Permutation von n Objekten berücksichtigen muss, hat mindestens einen Pfad (von der Wurzel zu einem Blatt) der Länge $O(n \log n)$. Also muss jeder vergleichende Sortieralgorithmus, der immer funktioniert, mindestens $O(n \log n)$ Rechenschritte durchführen. Damit wird verständlich, dass die Laufzeiten von Merge Sort und Quick Sort (im durchschnittlichen Fall) nicht wesentlich verbessert werden können, wenn man das Konzept des paarweisen Vergleichens beibehalten will.

Bemerkung 36: *Es gibt auch andere Sortierverfahren wie z.B. Counting Sort, Radix Sort und Bucket Sort, die mit anderen Prinzipien arbeiten und die in der Zeit $O(n)$ arbeiten.*

3. Dynamische Datenstrukturen

3.1. Lineare dynamische Datenstrukturen

Bis jetzt haben wir uns mit Algorithmen beschäftigt. Dabei kamen einzelne Variablen und Arrays fester Größe und verschiedenen Typs zum Einsatz. Allerdings werden in der Praxis oft kompliziertere Datenstrukturen benötigt, deren Größe nicht wie die eines Arrays (in C) festgelegt ist.

3.1.1. Einige Grundlagen der dynamischen Speicherverwaltung von C und C++

Zeiger geben den Ort / die Adresse eines Objekts im Speicher an (Analogie: Häuser $\triangleq$ Objekt im Speicher und Hausnummer $\triangleq$ Adresse). Die Standardbibliothek von C bietet Funktionen an,

mit der zusätzliche Objekte reserviert (z.B. mit `malloc`) und wieder frei gegeben (mit `free`) werden können.

Für einen mit `float *fp;` vereinbarten Zeiger reserviert `fp = malloc(sizeof(float));` den benötigten Speicher. Der Parameter von `malloc` gibt an, wieviele *Bytes* reserviert werden sollen, und das Resultat ist ein Zeiger auf das reservierte Speicherobjekt (Erfolgsfall) oder `NULL` (Fehlerfall). Angenommen `malloc` hat uns die Speicherstelle 17 reserviert, dann gilt `fp == 17` und `*fp = 3.0;` legt den Wert 3.0 im Kästchen 17 ab (siehe Abbildung 12). Der Operator `*`, angewendet auf einen Zeiger, ergibt den Inhalt des Speicherobjekts auf den der Zeiger deutet.

Nicht mehr benötigter Speicher *muss* mit `free` freigegeben werden. Ein ungültiger Zeiger, der auf keine Speicherstelle deutet, hat den Wert `NULL`.

Ein Zeiger kann nacheinander auf verschiedene Objekte im Speicher deuten (Zeigervariable). Der Zugriff auf ein dynamisch reserviertes (= allokiertes) Objekt ist nur via Zeiger möglich. Achtung: Nie darf ein Objekt noch reserviert sein, ohne das ein Zeiger auf das Objekt deutet (⇒ *Memoryleak*, da der Speicher nicht mehr freigegeben werden kann).

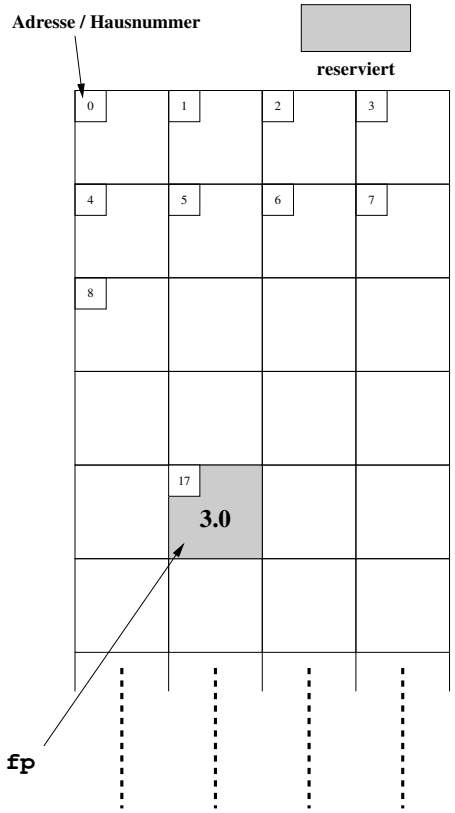


Abbildung 12: Speicherorganisation im Modell

3.1.2. Stapel

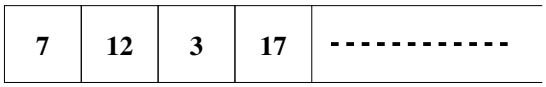
Ein Stapel ist eine Datenstruktur, die auch als Keller, Stack oder LIFO (= Last In First Out) bekannt ist. Ein Stapel kann, ähnlich wie ein Array, nur Elemente eines bestimmten Typs speichern (z.B. ein `int`-Stapel). Dabei ist der Zugriff auf die Elemente (z.B. über einen Index) eingeschränkt und folgt dem Konzept eines Papierstapels: Blätter können *oben* abgelegt und heruntergenommen werden. Es ist nicht möglich, in der Mitte etwas herausnehmen oder einschieben. Mit dieser Definition wird auch der Name LIFO klar.

Sei `T` der Typ der Objekte, die in einem Stack gespeichert werden. Dann sind die folgenden Operationen möglich:

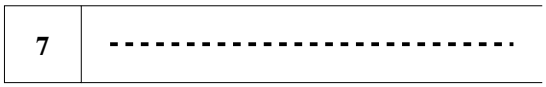
- `bool is_empty(stack<T> &s); /* true gdw. Stack leer */`
- `(void) push(stack<T> &s, T element); /* Lege element auf Stack s ab */`
- `T pop(stack<T> &s); /* Oberstes Element entfernen und zurückgeben */`
- `T top(stack<T> &s); /* Oberstes Element abfragen aber nicht entfernen */`

Beispiel 37: *Sei s ein Stack von* `ints`*, also* `stack<int> s;` *und* `s` *enthält zu Beginn kein Element:*

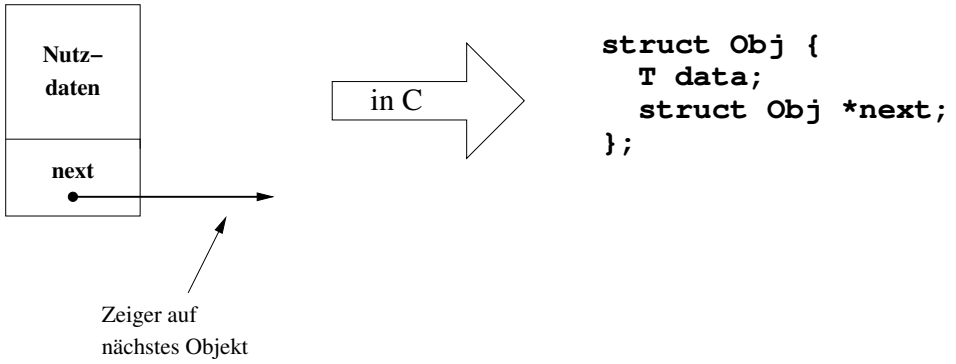
Nach `push(s,7); push(s,12); push(s,3); push(s,17);` *sieht der Stack wie folgt aus:*



Und drei `Pop`*-Operationen ergeben* 17, 3 *und* 12 *(in dieser Reihenfolge). Übrig bleibt:*



Mit Feldern können wir sicherlich leicht einen Stapel fester Größe implementieren. Aber: Wie machen wir das dynamisch? Dazu benutzen wir Objekte der folgenden Struktur:



Ablauf der Push-Operation

1. Neues Objekt reservieren mit `ptr = malloc(sizeof(struct Obj));` (und testen ob geklappt)
2. Speichern der Daten „wert“ in dem neuen Objekt mit `ptr -> data = wert; /* Analog: *ptr.data = wert; */`
3. Einketten als neues head-Objekt mit `ptr -> next = head;`
4. head-Zeiger neu setzen `head = ptr; /* ptr wird nicht mehr gebraucht */`

Eine graphische Darstellung der Push-Operation zeigt Abbildung 13 auf Seite 36.

Ablauf der Pop-Operation

1. Zeiger auf head-Objekt merken `ptr = head;`
2. Head auf Nachfolger setzen `head = head -> next;`
3. Daten auslesen `returnValue = ptr -> data;`
4. Speicher des entfernten Objekts freigeben `free(ptr);`

Eine graphische Darstellung der Pop-Operation zeigt Abbildung 14 auf Seite 36.

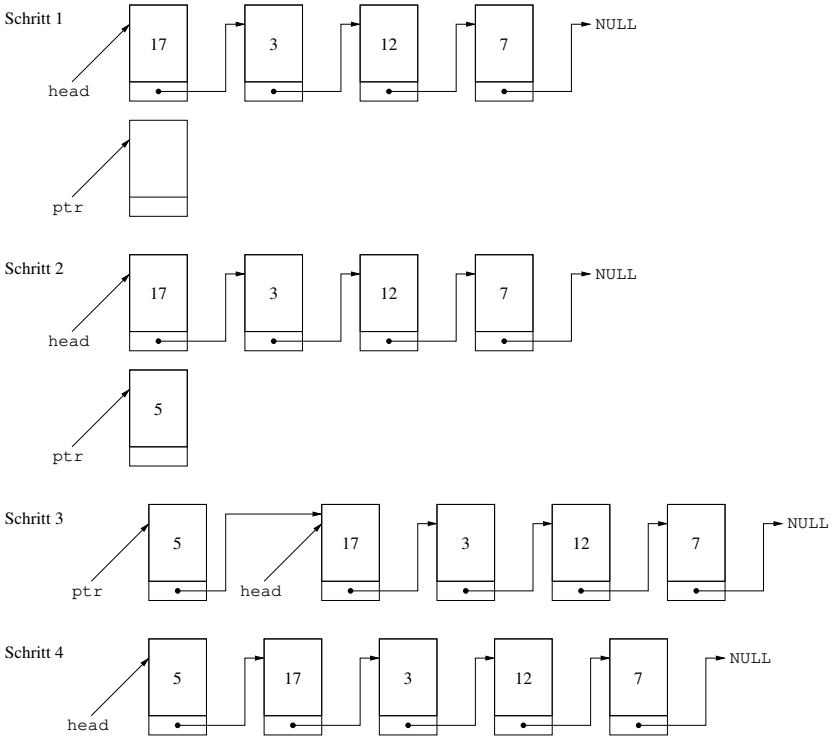


Abbildung 13: Ablauf der Push-Operation

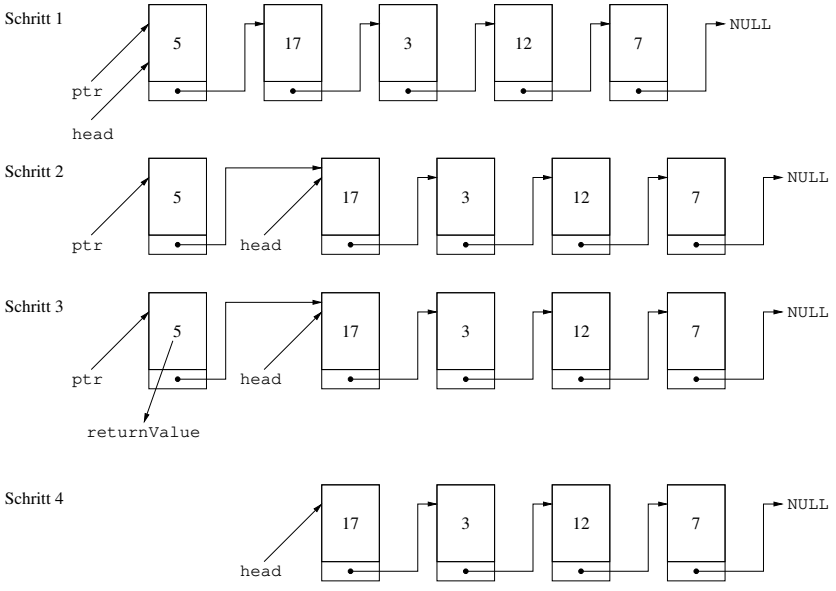


Abbildung 14: Ablauf der Pop-Operation

3.1.3. Der Stapel als abstrakter Datentyp

Die technische Realisierung (mit Zeigern) interessiert einen Anwender unseres Stacks nicht. D.h. wir „verstecken“ alle Interna (Geheimnisprinzip) des Stacks und der Anwender benutzt nur `push`, `pop` und `is_empty`. Das ist völlig analog zu den C Standarddatentypen wie `int` oder `float`.

Definition 38: *Wird ein Datentyp durch die anwendbaren Operationen bestimmt und damit das Verhalten und* nicht *die Realisierung festgelegt, dann spricht man von einem* abstrakten Datentyp *(kurz:* ADT*).*

In C könnte die Schnittstelle eines Stacks als ADT wir folgt aussehen:

```
/* Initialisiere den Stapel                                          */
/* Rückgabewert: 0 falls Stapel nicht angelegt werden konnte, 1 sonst */
int init();

/* Lege den Wert auf den Stapel */
/* Rückgabe: 0 falls nicht geklappt, 1 sonst */
int push(int data);

/* Hole den obersten Wert vom Stapel */
int pop();

/* Teste ob Stapel leer */
/* Rückgabewert: 1 falls Stapel leer, 0 sonst */
int is_empty();
```

Von der Realisierung mit Zeigern ist nun nichts mehr zu sehen. Auf den Stapel kann man nun nur noch durch die definierten Operationen zugreifen und alle *Implementationsdetails sind „geheim“* (Geheimnisprinzip für ADT). In C erreicht man dies, indem man alle Operationen in einer Header-Datei kapselt und die eigentlichen Implementierungsdetails in den `*.c`-Files unterbringt.

Bemerkung 39:

- *Bei der oben angegebenen, einfachen C-Implementierung, kann man nur eine Variable eines ADTs benutzen.*
  Ziel: Es sollen mehrere Variablen eines ADTs innerhalb eines Programms nutzbar sein.
- *Die Programmlogik eines Stapels von* `ints` *unterscheidet sich nicht von der eines Stapels von Kundendatensätzen, d.h. der Stapel sollte unabhängig von den zu verwaltenden Daten implementiert werden. Eine solche Implementierung ist bekannt als* generischer Datentyp*. Die Sprachmittel von C++ ermöglichen die Implementierung von generischen Datentypen (vgl. templates) besonders gut (vgl. die STL - Standard Template Library von C++).*

Für die Zeitkomplexität eines abstrakten Stapels ergibt sich:

Operation	Zeitkomplexität
`is_empty`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$

Da man nur Zugriff auf das oberste Element hat, dauern Zugriffe auf weiter unten liegende Elemente länger, d.h. das Suchen in n Elementen dauert im worst-case $O(n)$. Mögliche Anwendungen:

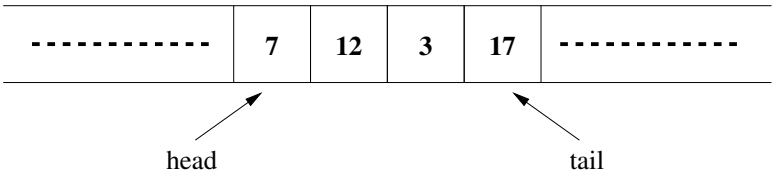
- Speicherung lokaler Variablen von Unterprogrammen in höheren Programmiersprachen (Der Compiler verwaltet hierzu einen Stapel im Hauptspeicher).
- Speichern von Rücksprungadressen bei Unterprogrammaufrufen und bei Interrupts.

3.1.4. Warteschlangen

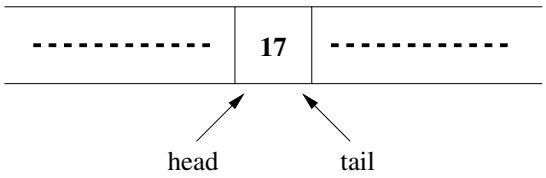
Eine Warteschlange, Queue oder auch FIFO-Speicher (First In First Out) simuliert das Warten in einer Schlange. Hinten werden Objekte an die Schlange angefügt und vorne werden Objekte wieder aus der Schlange entfernt. Dabei sind folgende Operationen zulässig:

- `bool is_empty(queue<T> &q); /* true gdw. Warteschlange leer */`
- `(void) append(queue<T> &q, T e); /* Füge e in die Queue q ein */`
- `T get(queue<T> &q); /* Entferne ein Element aus der Queue */`

Beispiel 40: *Sei q eine Warteschlange von* `ints`*, also* `queue<int> q;`*, und* `q` *enthält zu Beginn kein Element. Nach* `append(q,7); append(q,12); append(q,3); append(q,17);` *sieht die Warteschlange wie folgt aus:*



Und drei `get`*-Operationen ergeben* 7*,* 12 *und* 3 *(in dieser Reihenfolge). Übrig bleibt:*



Ablauf der Append-Operation

1. Ein neues Objekt reservieren mit `ptr = malloc(sizeof(Obj));` (und testen ob geklappt)
2. Speichern der Daten „wert“ in dem neuen Objekt `ptr -> data = wert;`.
3. Nachfolger von `ptr` initialisieren `ptr -> next = NULL;`
4. Neues Objekt an `tail` anhängen `tail -> next = ptr;`
5. `tail` neu setzen `tail = ptr; /* ptr wird nicht mehr gebraucht */`

Eine graphische Darstellung der Append-Operation zeigt Abbildung 15 auf Seite 39.

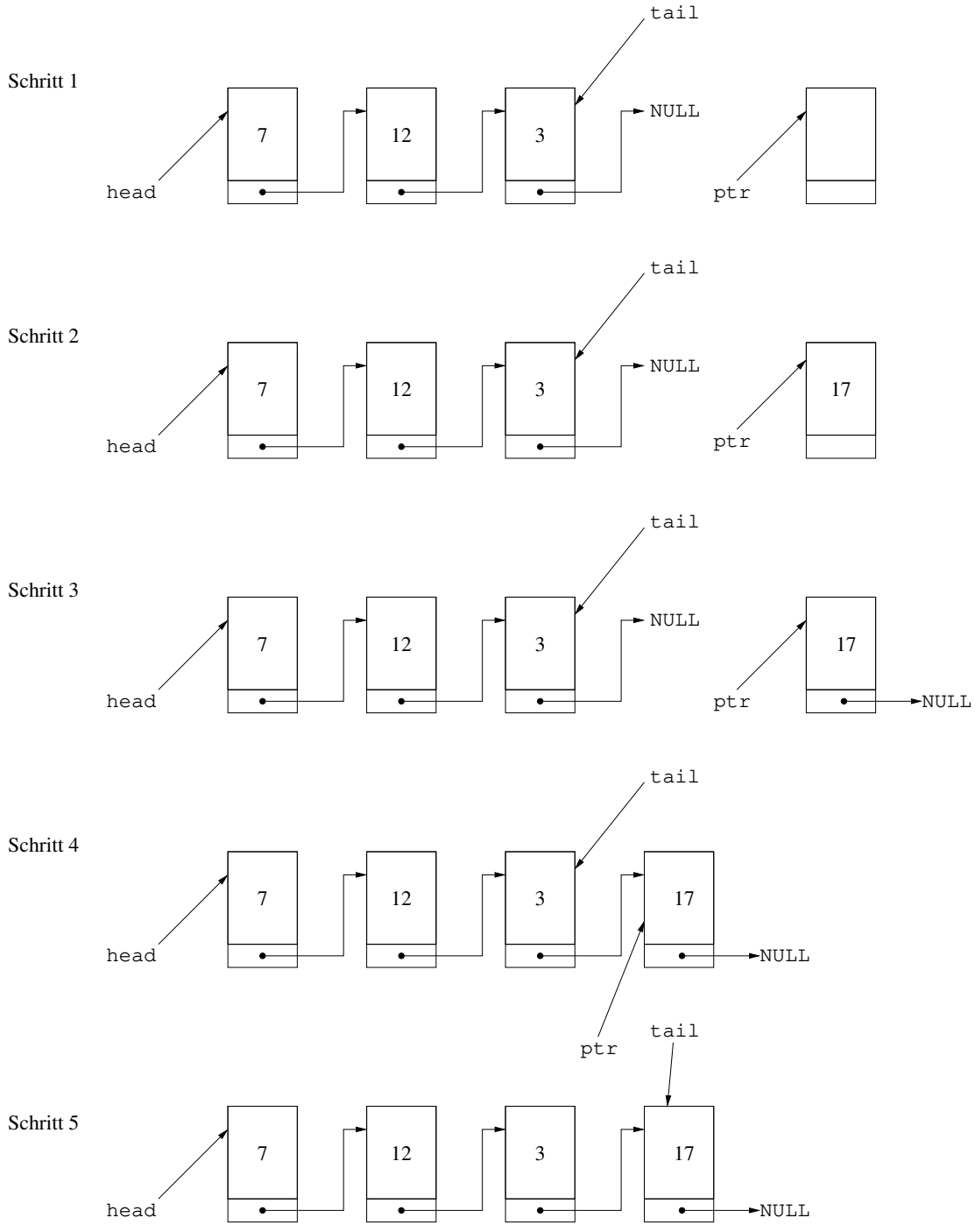


Abbildung 15: Ablauf der Append-Operation

Ablauf der Get-Operation

1. `head` kopieren `ptr = head;`
2. Informationen kopieren `returnValue = ptr -> data;`
3. `head` neu setzen `head = head -> next;`
4. Überflüssiges Element freigeben `free(ptr);`

Graphische Darstellung der Get-Operation: Übung

Bemerkung 41: *Die Verkettungsrichtung ist wichtig. Wir ketten beim Schwanz der Warteschlange neue Element ein und beim Kopf alte Elemente wieder aus. Angenommen die Verkettungsrichtung „ältere Elemente deuten auf neuere Elemente“ wäre umgekehrt, dann wäre die Append-Operation einfach (`tail` neu setzen), aber ausketten am Kopf wäre schwer, da wir keinen Zeiger auf das Vorgängerelement haben. ⇒ komplette Warteschlange durchlaufen um den Vorgänger zu finden.*

Für die Zeitkomplexität der Operationen einer abstrakten Warteschlange ergibt sich:

Operation	Zeitkomplexität
`is_empty`	$O(1)$
`append`	$O(1)$
`get`	$O(1)$

Mögliche Anwendungen von Warteschlangen:

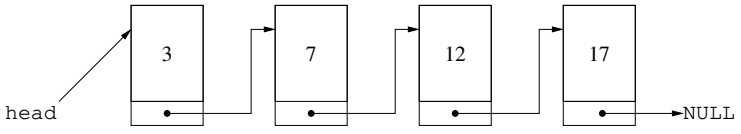
- Warteschlange von Prozessen in Betriebssystemen (vgl. z.B. `kfifo.c` im Linux-Kern)
- Warteschlange von Druckaufträgen

3.1.5. Lineare Listen

Bis jetzt war es nur möglich „vorne“ und/oder „hinten“ Elemente ein- bzw. auszuketten. Damit können wir Daten nicht geordnet verwalten (z.B. sollen Daten immer sortiert in einer Liste verwaltet werden).
Zulässige Operationen:

- `bool is_empty(list<T> &l); /* true gdw. Liste leer */`
- `(void) insert(list<T> &l, T i); /* Füge i in die sortierte Liste ein */`
- `T remove(list<T> &l, T o); /* Entferne o aus der Liste l */`

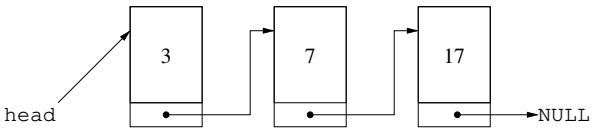
Angenommen wir haben die folgende sortierte Liste gegeben:



Ablauf der Remove-Operation
Es soll `remove(12)` simuliert werden:

1. Setze „Vorgänger“ auf `head`.
2. Rücke den „Vorgänger“ so lange vor, bis er auf ein Objekt deutet, dessen `next`-Zeiger auf das Objekt deutet, das die zu entfernenden Daten enthält.
3. Sichere den Zeiger auf den direkten Nachfolger des „Vorgängerobjekts“ in den temporären Zeiger `tmpPtr`.
4. Setze den `next`-Zeiger des „Vorgängersobjekts“ auf den `next`-Zeiger von `*tmpPtr`.
5. Lösche das überflüssige Objekt `free(tmpPtr);`.

Wir erhalten:

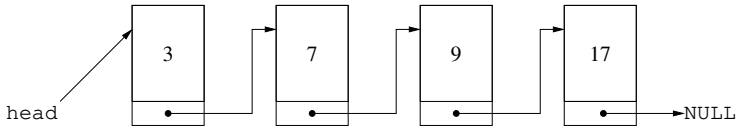


Eine graphische Darstellung der Remove-Operation zeigt Abbildung 16 auf Seite 42

Ablauf der Insert-Operation
Es soll `insert(9)` simuliert werden:

1. Neues Speicherobjekt erzeugen (mit `malloc`) und Daten in diesem Objekt speichern mit `tmpPtr = malloc(sizeof(Obj)); tmpPtr -> data = 9;`.
2. „Vorgänger“ auf den Anfang der Liste setzen.
3. „Vorgänger“ solange vorrücken, bis er auf das Objekt deutet, nach dem das neue Objekt einsortiert werden soll.
4. `next`-Zeiger von `*tmpPtr` auf den Nachfolger des Vorgängerobjekts zeigen lassen.
5. next-Zeiger des Vorgängerobjekts auf das im ersten Schritt erzeugte Objekt deuten lassen `vorgaenger -> next = tmpPtr;`.

Wir erhalten:



Eine graphische Darstellung der Insert-Operation zeigt Abbildung 17 auf Seite 43.

Bemerkung 42: *Die oben beschriebenen Insert- und Remove-Operation funktionieren nur im „inneren Teil“ der Liste. Sollten diese Operationen den Anfang oder das Ende der Liste betreffen, so müsste dies als Sonderfall ausprogrammiert werden. Dies ist fehleranfällig und arbeitsintensiv. Idee: Füge ein „kleinstes“ und ein „größtes“ Dummyelement ein, die beide ungültige Daten enthalten.*

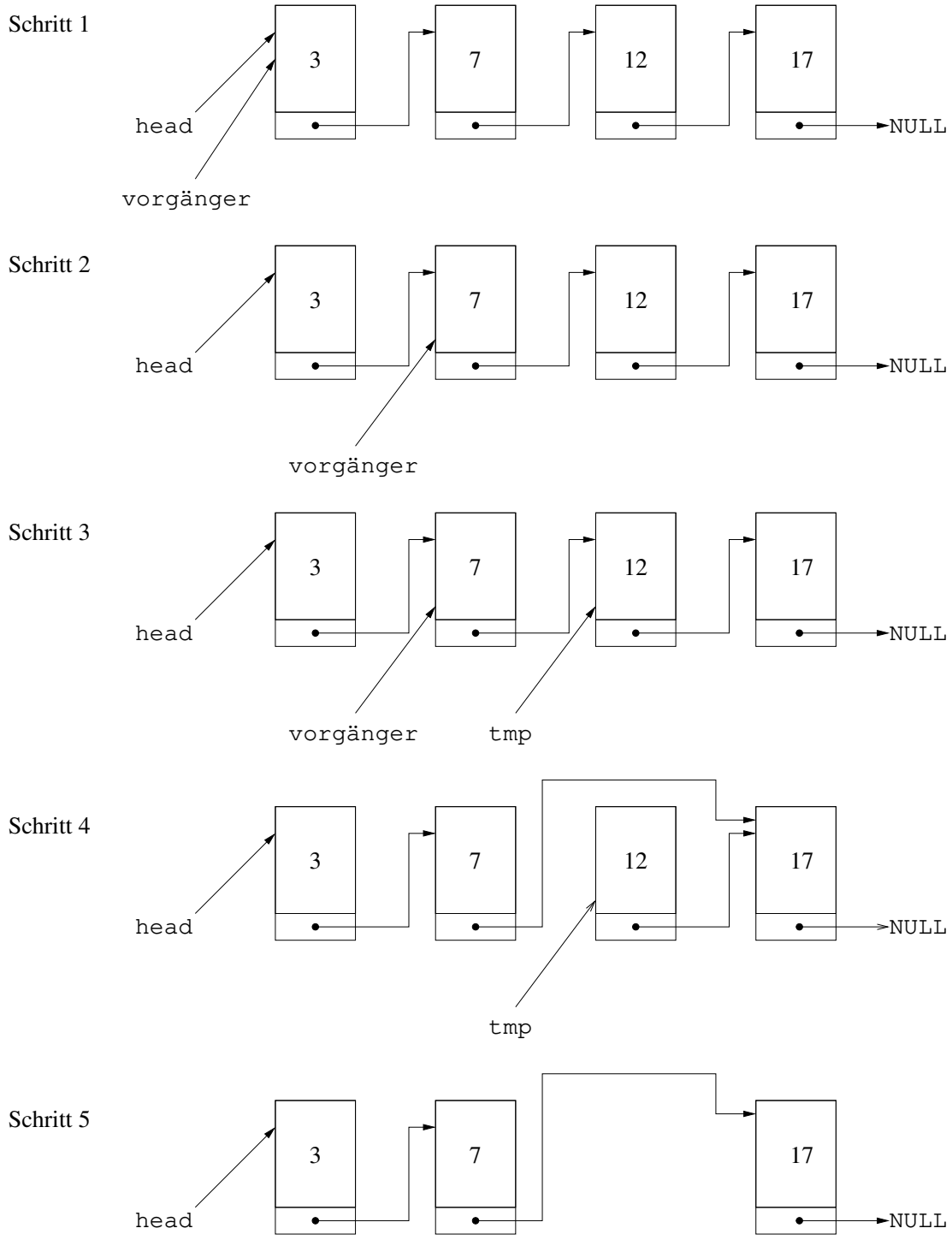


Abbildung 16: Ablauf der Remove-Operation

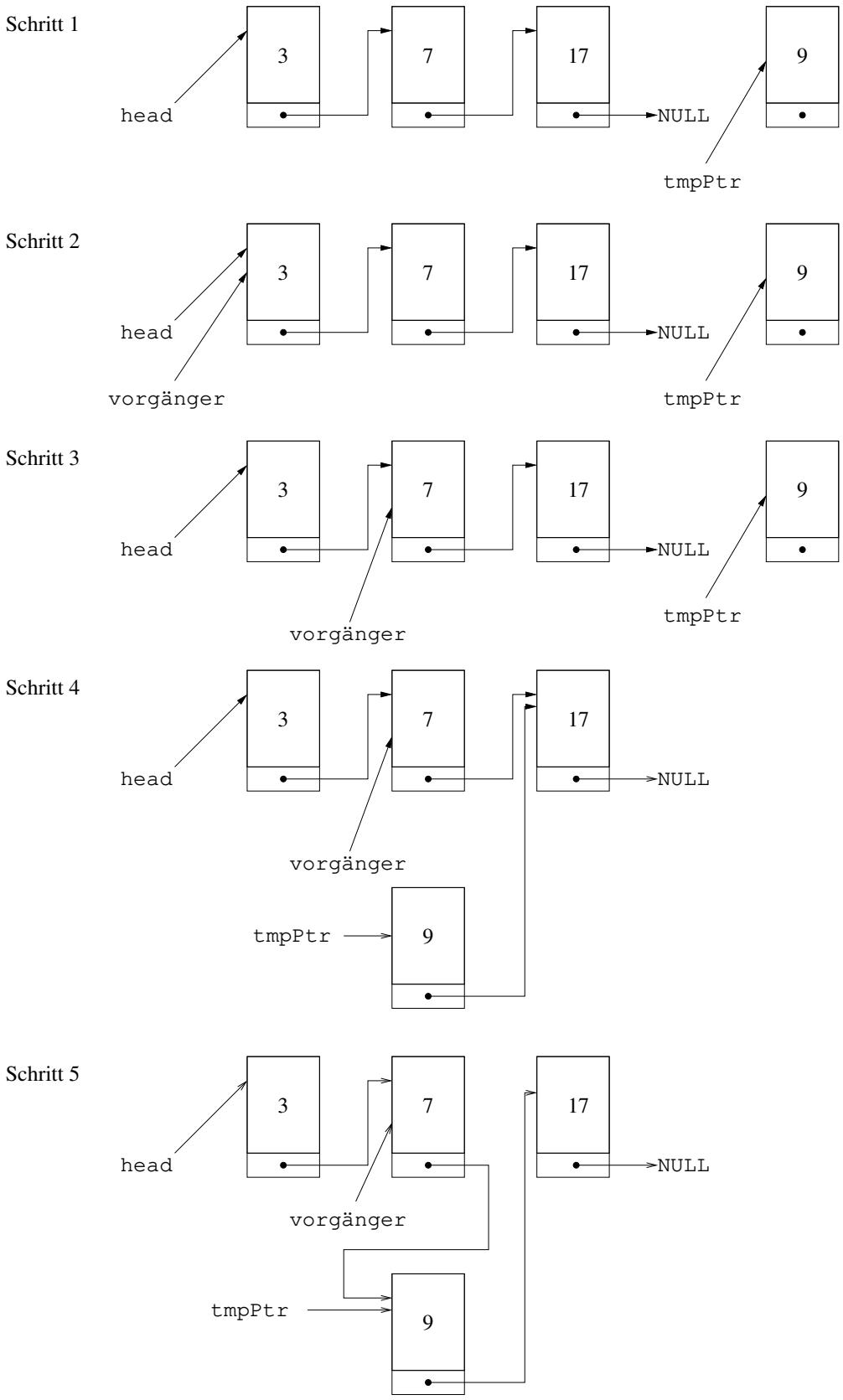
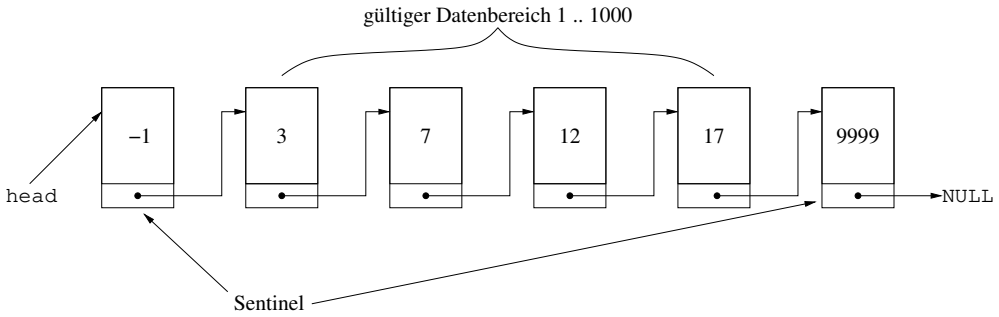


Abbildung 17: Ablauf der Insert-Operation



Diese Dummyelemente werden auch Sentinel *(engl. Wächter / Wachposten) genannt. Mit Sentinels werden die Spezialfälle bei* `insert` *und* `remove` *vermieden, da nur noch im „inneren Teil“ der Liste gearbeitet wird.*

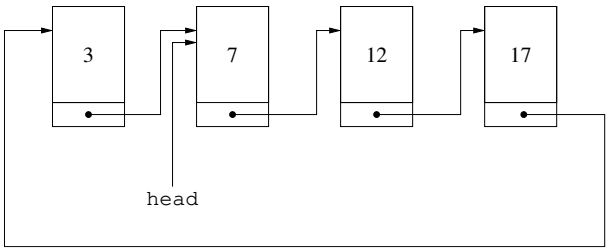
Die Zeitkomplexität für lineare Listen:

Operation	Zeitkomplexität
`is_empty`	$O(1)$
`insert`	$O(n)$
`remove`	$O(n)$

Bemerkung 43: *Lineare Listen kann man mit den schon vorgestellten Algorithmen sortieren. Evtl. dauert das länger, weil man nicht direkt auf ein Listenelement zugreifen kann.*
Übung: Programmieren Sie eine Swap-Operation von zwei aufeinander folgenden Listenelementen.

3.1.6. Weitere lineare Datenstrukturen

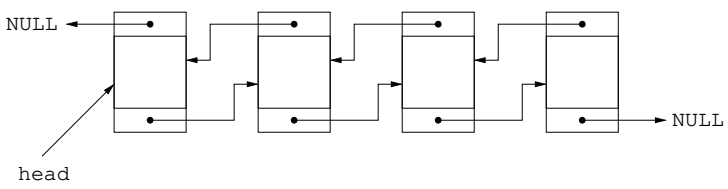
Man kann eine lineare Liste auch in eine Ringliste umwandeln:



Anwendungen:

- Zuteilung von Zeitscheiben in einem Betriebssystem
- Sende- und Empfangsbuffer bei der Datenkommunikation

Ein Nachteil bei den linearen Listen war, dass man nur in einer Richtung vorrücken konnte (mit Hilfe der `next`-Zeiger). Dies kann man ausgleichen, wenn man Verkettungen in zwei Richtungen einführt:



3.1.7. Kombination von statischen und dynamischen Ansätzen

Will man große Datensätze (z.B. vollständige Personaldatensätze) speichern, so kann man

- den Datensatz in einem dynamisch erzeugten Objekt anlegen und
- einen Schlüssel (z.B. die Personalnummer) und einen Zeiger auf das dazugehörige dynamische Objekt in einem statischen Feld ablegen.

Vorteile gegenüber einer rein dynamischen Lösung

- direkter Zugriff über den Schlüssel
- schnelles Suchen bzgl. des Schlüssels

Vorteile gegenüber einer rein statischen Lösung

- wenig Speicherverschwendung, wenn nur wenige Datensätze gespeichert werden
- große Sortiergeschwindigkeit (die Datensätze selbst müssen nicht bewegt werden)

3.2. Bäume

Bisher haben wir nur lineare Datenstrukturen angesehen. Nun wollen wir eine Familie von Datenstrukturen untersuchen, die uns die Darstellung und Speicherung von hierarchischen Strukturen ermöglicht (vgl. Stammbaum).

Definition 44: *Das Paar* (V, E) *mit* $E \subseteq V \times V$ *ist ein* Baum*, wobei* V *die (endliche) Menge der* Knoten *und* E *die Menge der* gerichteten Kanten *zwischen den Knoten ist, wenn*

- *es gibt einen ausgezeichneten Knoten* w*, die* Wurzel*, und*
- *jeder Knoten* k *(auch* Kind *von* v *genannt), der nicht die Wurzel ist, ist genau mit* einer *Kante mit seinem* Vaterknoten v *verbunden.*

Ein Knoten ohne Kinder heißt Blatt*, alle anderen Knoten bezeichnet man als* innere Knoten*. Ein* Pfad *ist eine Folge von Knoten, die durch Kanten verbunden sind. Zwischen jedem Knoten und der Wurzel gibt es genau* einen *Pfad. Die* Länge *eines Pfades ist die Anzahl der Kanten im Pfad. Die* Höhe *eines Baums ist die Anzahl der Knoten im längsten Pfad. Alle Knoten, die von der Wurzel einen Pfad der Länge* h *haben, bilden die* Ebene h*. Ist die Anzahl der Kinder jedes Knotens* $\leq n$*, dann spricht man von einem* n*-nären Baum.*

Mögliche Anwendungen:

- Anordnung von Dateien in einem Dateisystem
- Darstellung von arithmetischen Ausdrücken (siehe Abbildung 19 auf Seite 46)
- Gliederungen von Menüs
- Heapsort und Prioritätswarteschlangen

Eine häufig in der Praxis verwendete Klasse von Bäumen sind die *Binärbäume*.

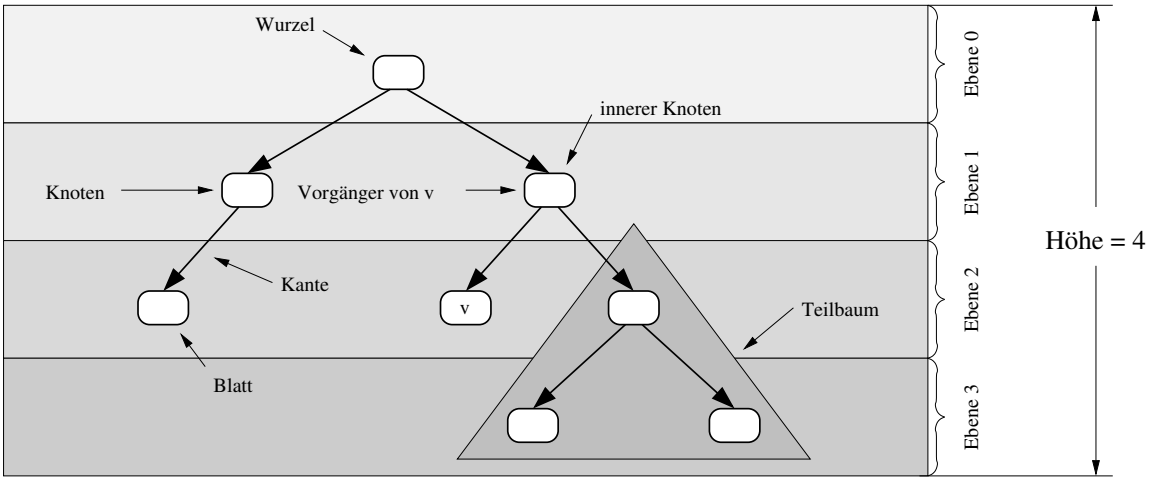


Abbildung 18: Beispiel für einen Binärbaum

((3 + 4) * 5) + (2 * 3)

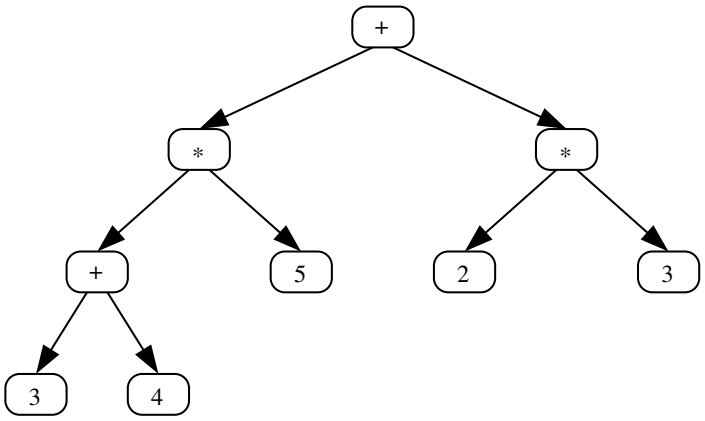
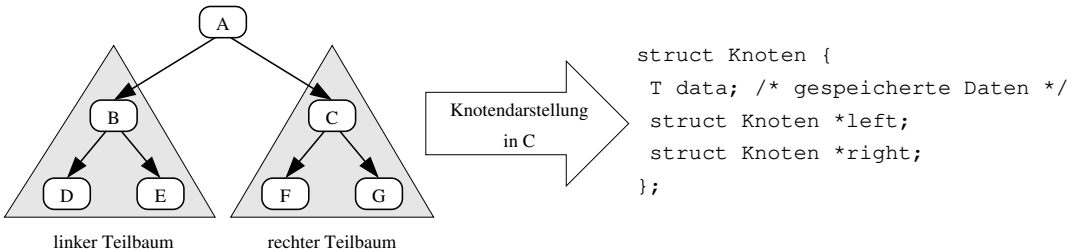


Abbildung 19: Ein arithmetischer Ausdruck in Baumform

3.2.1. Algorithmen zur Traversierung

Oft wird eine systematische Art des Durchlaufens aller Knoten eines Baums benötigt. Dieser Vorgang ist unter *Traversierung* (engl. traversal) bekannt. Es wird sich zeigen, dass viele Algorithmen für (Binär)bäume Verallgemeinerungen dieser Basisalgorithmen sind. Für die weiteren Beispiele sei der folgenden Binärbaum gegeben:




```
struct Knoten {
 T data; /* gespeicherte Daten */
 struct Knoten *left;
 struct Knoten *right;
};
```

Inorder-Durchlauf

- Besuche den linken Teilbaum
- Besuche den Knoten selbst
- Besuche den rechten Teilbaum

Für unseren Baum ergibt sich die Reihenfolge: D B E A F C G
Für einen Inorder-Durchlauf verwenden wir den folgenden rekursiven Algorithmus:

Eingabe: Wurzel t eines Binärbaums
Ergebnis: Verarbeitung der Knoten in Inorder

```
if (t != NULL) then
    /* Verarbeite den linken Teilbaum */
    Inorder(t.left);
    Verarbeite t.data;
    /* Verarbeite den rechten Teilbaum */
    Inorder(t.right);
end
```

Algorithmus 15: Inorder

Preorder-Durchlauf

- Besuche den Knoten selbst
- Besuche den linken Teilbaum
- Besuche den rechten Teilbaum

Für unseren Baum ergibt sich die Reihenfolge: A B D E C F G

Postorder-Durchlauf

- Besuche den linken Teilbaum
- Besuche den rechten Teilbaum
- Besuche den Knoten selbst

Für unseren Baum ergibt sich die Reihenfolge: D E B F G C A

Bemerkung 45: *Durchläuft man die Baumdarstellung eines arithmetischen Ausdrucks Inorder, so ergibt sich die normale Darstellung des Ausdrucks, wogegen ein Postorder-Durchlauf den Ausdruck in umgekehrt polnischer Notation ergibt.*

3.2.2. Suchbäume

Bisher haben wir mit Bäumen Daten hierarchisch organisiert. Nun wollen wir Bäume dazu verwenden effizient in Daten zu suchen. Dazu speichern wir in jedem Knoten zusätzlich noch einen *Schlüsselwert*. Solche Datenstrukturen nennt man auch *Wörterbücher* oder *Dictionaries*. Im folgenden beschränken wir und auf binäre Suchbäume, die aus dem Knotentyp

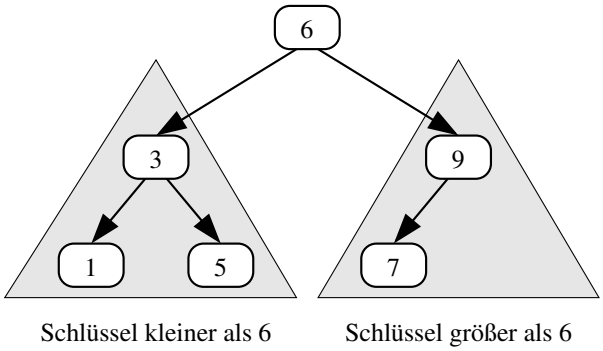


Abbildung 20: Beispiel für einen Suchbaum

```
struct Knoten {
 S key; /* Schlüssel des Knotens */
 T data; /* Daten */
 struct Knoten *left; /* Linker Teilbaum */
 struct Knoten *right; /* Rechter Teilbaum */
}
```

aufgebaut sind und die die folgenden Eigenschaften haben:

- Der Knoten `k` enthält den Schlüssel `k.key`
- Alle Schlüsselwerte im linken Teilbaum `k.left` sind kleiner als `k.key`
- Alle Schlüsselwerte im rechten Teilbaum `k.right` sind größer als `k.key`

Suchen in Suchbäumen Damit ergibt sich der folgende Algorithmus um in Suchbäumen zu suchen:

Eingabe: Zeiger auf den Wurzelknoten k des Suchbaums, Schlüssel x
Ergebnis: Zeiger auf den Knoten der x enthält oder `NULL`, wenn Schlüssel nicht gefunden

if *(k == NULL)* **then**
  return `NULL`;
else if *((k -> key) < x)* **then**
  return SearchTree(k -> right, x);
else if *((k -> key) > x)* **then**
  return SearchTree(k -> left, x);
else
  return k;
end

Algorithmus 16: SearchTree

Übung: Formulieren Sie eine iterative Variante von `SearchTree`.

Einfügen in Suchbäumen Beim Einfügen müssen die Suchbaumeigenschaften erhalten bleiben, d.h. wir müssen nach der korrekten Einfügeposition suchen. Dabei müssen wir zwei Fälle unterscheiden:

- Der Baum ist leer, dann wird der einzufügende Knoten zur Wurzel des Baums
- Die Baum enthält einen Knoten, dann müssen wir nach dem Knoten suchen, der der Elternknoten des einzufügenden Knotens wird.

Eingabe: Zeiger auf den Wurzelknoten k des Suchbaums, Schlüssel x
Ergebnis: Zeiger auf den Knoten der x enthält oder `NULL`, wenn Schlüssel schon enthalten

```
if (k == NULL) then
    /* Baum noch leer */
    erzeuge Knoten *w;
    w -> key = x;
    return w;
else
    return InsertTreeSub(k, x);
end
```

Algorithmus 17: InsertTree

Entfernen aus Suchbäumen Dies ist die komplizierteste Operation auf Suchbäumen. Dabei müssen wir drei Fälle unterscheiden:

1. Der zu entfernende Knoten k ist ein Blatt:
   - Merke den Zeiger auf den zu entfernenden Knoten k
   - Setze den passenden left- oder right-Zeiger des Elternknotens von k auf `NULL`
   - gib den für k reservierten Speicher frei
2. Der zu entfernende Knoten k ist kein Blatt und hat genau ein Kind (siehe Abbildung 21):
   - Merke den Zeiger auf den zu entfernenden Knoten k
   - Setze den entsprechenden left- oder right-Zeiger des Elternknotens v von k auf den Wert des passenden left- oder right-Zeigers von k.
   - gib den für k reservierten Speicher frei
3. Der zu entfernende Knoten k ist kein Blatt und hat genau zwei Kinder (vgl. Abbildung 22):
   - Merke den Zeiger auf den zu entfernenden Knoten k
   - Suche das größte Element g im linken Unterbaum B_2. Dieses ist das Blatt ganz rechts.
   - Lösche die Referenzen auf g aus B_2 heraus
   - Setze den passenden left- oder right-Zeiger des Elternknotens v von k auf g
   - Setze den left-Zeiger von g auf die Wurzel von B_2
   - Setze den right-Zeiger von g auf die Wurzel von B_3
   - gib den für k reservierten Speicher frei

Bemerkung 46: *Werden (teilweise) geordnete Daten in einen Suchbaum eingefügt, so entsteht ein entarteter Suchbaum (siehe Abbildung 23).*

Definition 47: *Wir nennen einen Baum* vollständig *der Höhe k, wenn alle Ebenen des Baums vollständig besetzt sind.*

Eingabe: Zeiger auf den Wurzelknoten k des Suchbaums (Baum enthält mindestens einen Knoten), Schlüssel x
Ergebnis: Zeiger auf den Knoten der x enthält oder `NULL`, wenn Schlüssel schon enthalten

```
act = k;
                                          /* Nach korrekter Position suchen */
while (true) do
    if ((act -> key) == x) then
                                                  /* Element schon im Baum */
        return NULL;
    else if ((act -> key) < x) then
        if (Blatt erreicht) then
            erzeuge Knoten *w;
            w -> key = x;
            act -> right = w;
            return w;
        else
                                                      /* Im Baum absteigen */
            act = act -> right;
        end
    else if ((act -> key) > x) then
        if (Blatt erreicht) then
            erzeuge Knoten *w;
            w -> key = x;
            act -> left = w;
            return w;
        else
                                                      /* Im Baum absteigen */
            act = act -> left;
        end
end
```

Algorithmus 18: InsertTreeSub

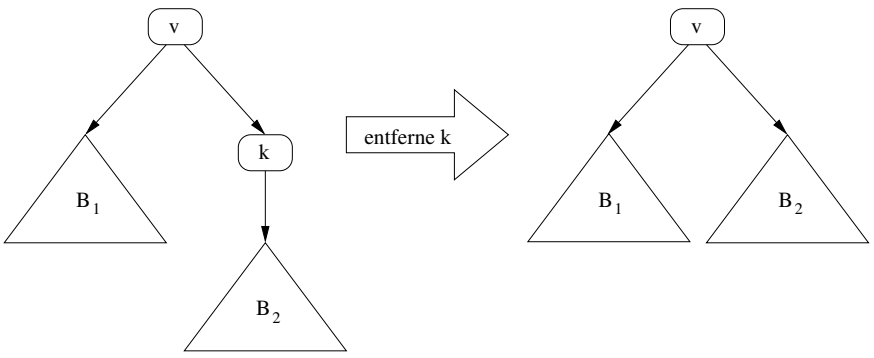


Abbildung 21: Entfernen eines Knoten mit einem Nachfolger

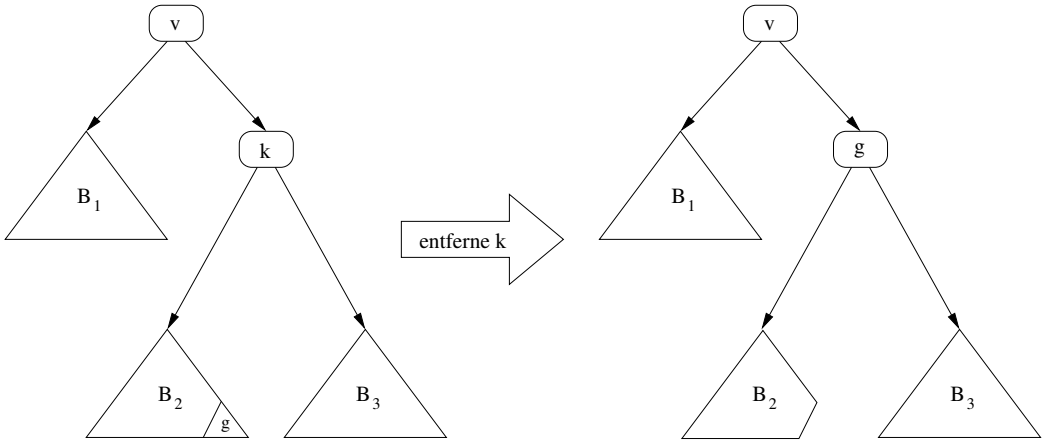


Abbildung 22: Entfernen eines Knoten mit zwei Nachfolgern

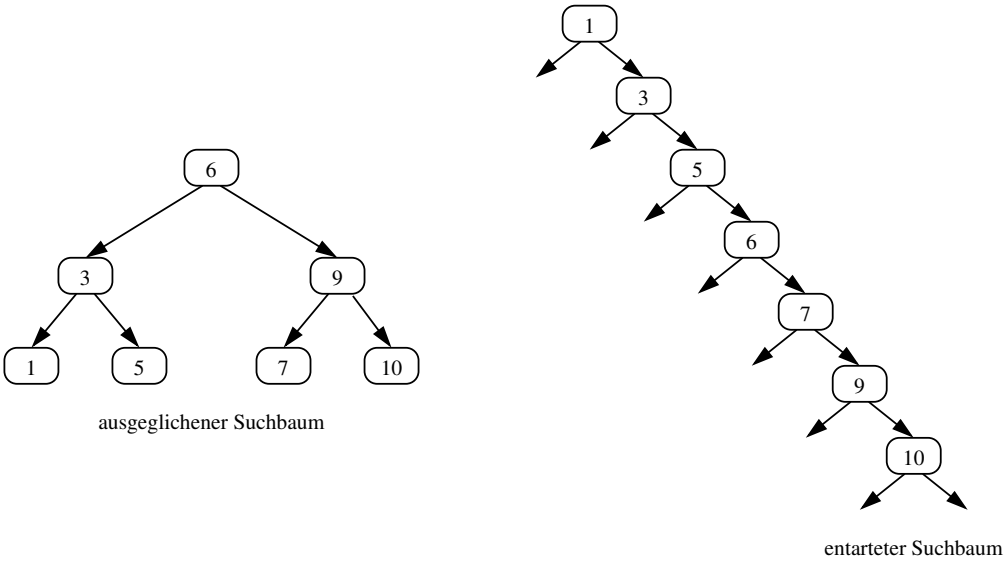


Abbildung 23: Beispiel für einen balancierten und einen entarteten Suchbaum

Lemma 48: *Ein vollständiger binärer Suchbaum der Höhe k hat $2^k - 1$ Knoten.*

Beweis: Wir zeigen das mit Hilfe einer vollständigen Induktion über die Baumhöhe:

(IA) Sei $k = 1$, dann ist $2^1 - 1 = 1$. Das stimmt, da ein binärer Suchbaum der Höhe 1 nur aus dem Wurzelknoten besteht.

(IV) Ein vollständiger binärer Suchbaum der Höhe k hat $2^k - 1$ Knoten.

(IS) $k \to k{+}1$: Ein vollständiger binärer Suchbaum der Höhe $k{+}1$ besteht aus dem Wurzelknoten und zwei Teilbäumen der Höhe k. D.h. wenden wir die (IV) an, so ergibt sich eine Knotenzahl von $2 \cdot \underbrace{(2^k - 1)}_{\text{Knoten eines Baums der Höhe } k} + 1 = 2 \cdot 2^k - 1 = 2^{k+1} - 1$, d.h. die Aussage ist korrekt. #

Aus Lemma 48 ergibt sich direkt, dass ein ausgeglichener (vollständiger) Suchbaum mit n Knoten eine Höhe von $O(\log n)$ hat. Es ergibt sich also die folgende Komplexität:

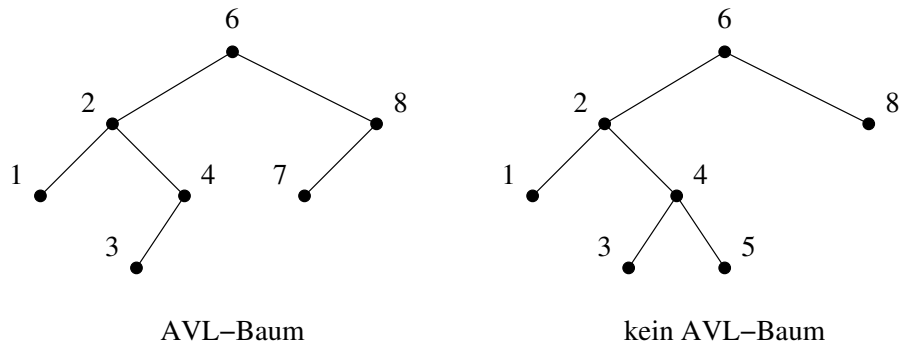


Abbildung 24: Ein Beispiel für das AVL-Kriterium

Operation	Arrays	Listen	Suchbaum (ausgeglichen)	Suchbaum (entartet)
Search	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
Insert	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
Remove	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$

Es stellt sich also die Frage, wie man das Entarten von Suchbäumen verhindern kann.

3.2.3. AVL-Bäume

Bei dem Einsatz von Suchbäumen trat immer wieder das Problem der Entartung auf. Dies hat zur Folge, dass Such- und Einfügeoperationen verhältnismäßig lange dauern können ($O(n)$ statt $O(\log n)$).

Idee: Versuche den Aufwand durch „Ausgleichen“ zu minimieren.

Eine Möglichkeit diese Idee umzusetzen sind die *AVL-Bäume*, die nach den russischen Mathematikern G. M. Adelson-Velskii und E. M. Landis benannt wurden. Um AVL-Bäume beschreiben zu können, benötigen wir die folgende Definition:

Definition 49 (AVL-Kriterium): *Ein Suchbaum ist ein* AVL-Baum*, wenn für* jeden *(inneren) Knoten der absolute Betrag der Differenz der Höhen des linken und des rechten Teilbaums maximal* 1 *ist.*

Ein Beispiel für diese Definition findet sich in Abbildung 24.

Bemerkung 50: *In Definition 49 ist es wichtig, dass für* jeden Knoten *der absolute Betrag der Differenz der Höhen der Teilbäume höchstens* 1 *ist. Dies lässt sich leicht durch Abbildung 25 erkennen.*

Vergleichbar wie bei den Suchbäumen, werden die Operationen Suchen, Einfügen und Löschen in AVL-Bäumen benötigt.

Suchen in AVL-Bäumen Da jeder AVL-Baum auch ein Suchbaum ist, können wir zum Suchen in AVL-Bäumen direkt Algorithmus 16 verwenden.

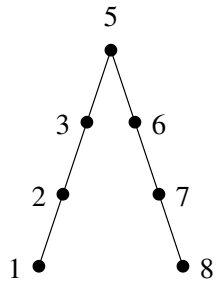


Abbildung 25: Ein nicht ausgeglichener Suchbaum

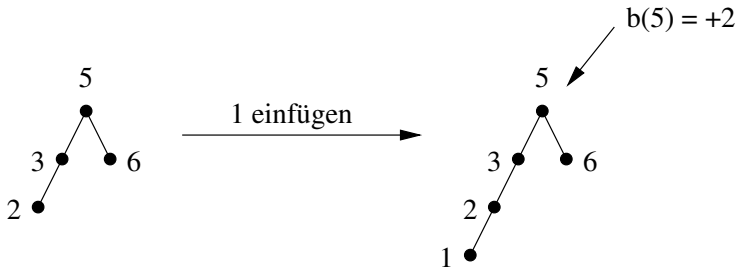


Abbildung 26: Balance nach Einfügen in einen AVL-Baum

Einfügen in AVL-Bäumen Fügt man einen neuen Knoten in einen AVL-Baum ein, so kann es passieren, dass der entstehende Baum kein AVL-Baum mehr ist. Aus diesem Grund muss der entstehende Baum nachträglich „repariert“ werden, um wieder einen AVL-Baum zu erhalten. Dazu benötigen wir die Definition der *Balance* eines Knotens:

Definition 51: *Die* Balance $b(k)$ *eines Knoten* k *ist definiert durch*

$$b(k) =_{\text{def}} \text{height}(k.\mathit{left}) - \text{height}(k.\mathit{right}).$$

Damit ist die Balance eines Knotens die Differenz der Höhen des linken und des rechten Teilbaums eines Knotens k*.*

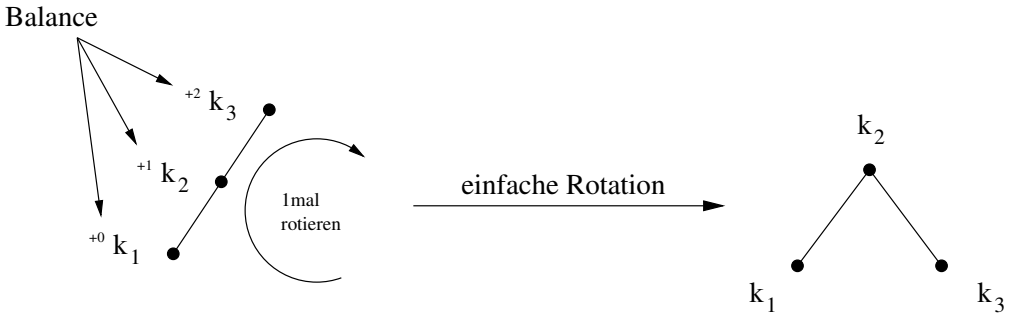
Für AVL-Bäume gilt die AVL-Bedingung: $b(k) \in \{-1, 0, 1\}$ (AVL-Bedingung)

Nach dem Einfügen eines Knotens in einen AVL-Baum kann aber auch gelten: $b(k) \in \{-2, -1, 0, 1, 2\}$ wie Abbildung 26 zeigt. Damit wird klar, dass nach einer Einfügeoperation Knoten vertauscht werden müssen, so dass die Balance jedes Knotens wieder die AVL-Bedingung erfüllt. Dabei muss darauf geachtet werden, dass die Suchbaumeigenschaft erhalten bleibt. Bei einer genaueren Untersuchung können vier verschiedene Fälle identifiziert werden, wobei sich schnell zeigt, dass je zwei dieser Fälle symmetrisch behandelt werden können. Folgende Fällen sind möglich:

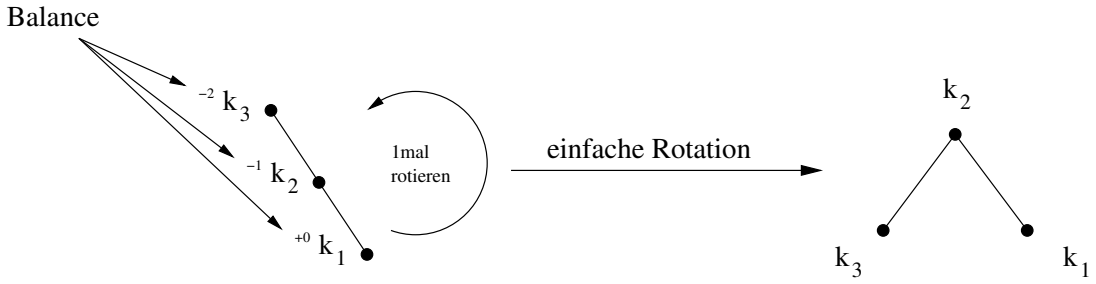
1. Einfügen in den linken Teilbaum des linken Kindes
2. Einfügen in den rechten Teilbaum des linken Kindes
3. Einfügen in den rechten Teilbaum des rechten Kindes

4. Einfügen in den linken Teilbaum des rechten Kindes

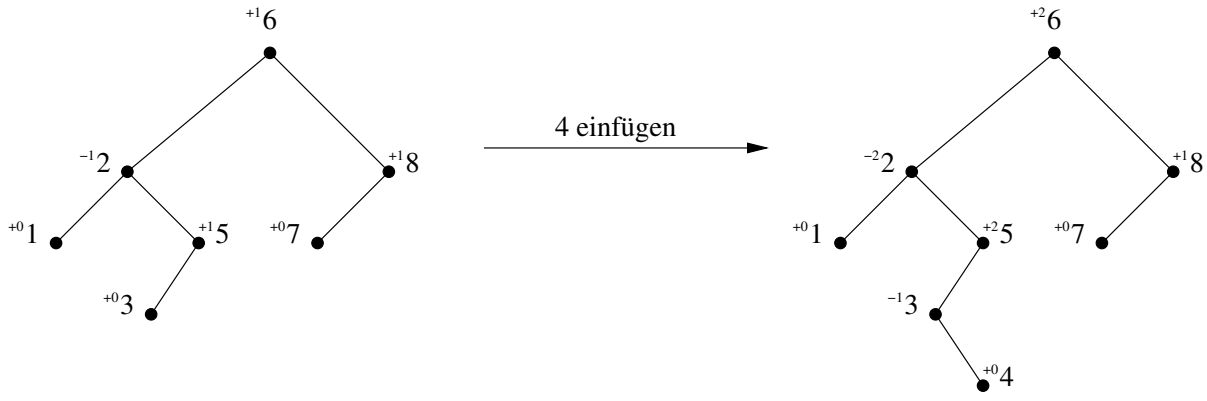
Im Fall 1 wurde ein Knoten in dem linken Teilbaum des linken Kindes eingefügt, d.h. wir suchen, von den Blättern beginnend, nach einem Knoten k_1, dessen Großvater als erstes die AVL-Bedingung verletzt. Diese Verletzung wird dann wie folgt repariert:



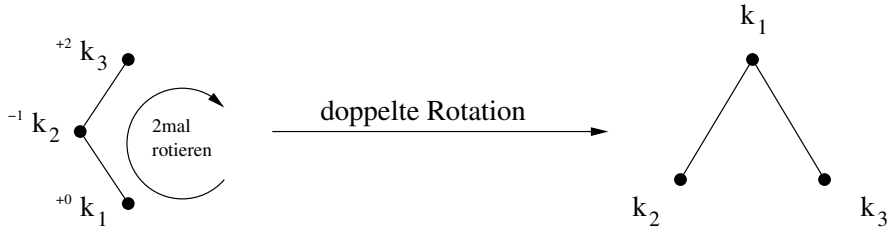
Fall 3 wird völlig analog behandelt, da er spiegelsymmetrisch zu Fall 1 ist.



Wurde eine neuer Knoten nach Fall 2 in einen AVL-Baum eingefügt, dann ergibt sich eine kompliziertere Situation. Dieser Fall ergibt sich z.B. bei der folgenden Einfügeoperation:



Für den allgemeinen Fall vom Typ 2 ergibt sich:



Wieder bleibt die Suchbaumeigenschaft erhalten, denn sowohl vor der Umformung als auch nach der doppelten Rotation gilt $k_1 < k_3$, $k_2 < k_3$ und $k_2 < k_1$, wie leicht nachgeprüft werden kann. Fall 4 kann nun völlig analog abgehandelt werden, da dieser zu Fall 2 spiegelsymmetrisch ist. Abschließend soll das nächste Beispiel die Einfügeoperation in AVL-Bäume nochmal verdeutlichen:

Beispiel 52: *Gegeben sei ein leerer AVL-Baum. Abbildung 27 zeigt die notwendigen Rotationen, die beim Einfügen von* $3, 2, 1, 4, 5, 6, 7, 16$ *und* 15 *(in dieser Reihenfolge) in einen AVL-Baum durchgeführt werden müssen.*

Entfernen aus AVL-Bäumen Das Vorgehen beim Löschen aus AVL-Bäumen ähnelt dem Vorgehen, dass beim Einfügen verwendet wurde. Wir löschen einen Knoten wie aus einem gewöhnlichen Suchbaum und korrigieren evtl. auftretende Ungleichgewichte durch die schon bekannten Rotationsoperationen. Dabei ist zu beachten, dass der nach einer Rotationsoperation entstehende Baum immer noch die AVL-Eigenschaft verletzen kann. D.h. eventuell sind weitere Rotationsoperationen notwendig, die durchgeführt werden müssen, bis der Wurzelknoten erreicht ist.

4. Hashverfahren

Bis jetzt wurden (effiziente) Datenstrukturen untersucht, die das Suchen und Einfügen in garantierter Zeit $O(n \log n)$ unterstützen. Das Ziel dieses Abschnittes ist es, eine neue Art von Datenstruktur, so genannte *Hashtables*, zu untersuchen, bei der die Einträge *fast* immer in konstanter Zeit gefunden, eingefügt und gelöscht werden können. Dabei sollen die Datensätze geeignet in einem Array gespeichert werden.

Dazu ist eine spezielle Funktion notwendig, die konkret zu speichernde Einträge / Datensätze so auf das Array verteilt (meist in konstanter Zeit), dass sie auch (fast) immer in konstanter Zeit gefunden werden kann. Solche Funktionen nennt man *Hashfunktionen.*

Das Grundprinzip des *Hashens* (engl. zerhacken, Prüfsumme):

- Die Speicherung erfolgt in einem Feld der Größe N, d.h. die Indexwerte sind $0, \ldots N-1$. Die einzelnen Komponenten des Feldes werden üblicherweise auch *Buckets* genannt.
- Eine Hashfunktion h bestimmt für einen Eintrag e den Index $h(e)$.
- h sorgt für eine „gute“ / „gleichmäßige“ Verteilung, d.h. es passiert sehr selten, dass zwei verschiedene Elemente e und e' an der gleichen Position $h(e) = h(e')$ gespeichert werden sollen (*Kollision*).

Anschaulich berechnet eine Hashfunktion einen „Fingerabdruck“ des Datensatzes e, denn ähnlich wie zwei Menschen unterschiedliche Fingerabdrücke haben, sind die Hashwerte von zwei Datensätzen (idealerweise) unterschiedlich.

Üblicherweise ist die Anzahl der verschiedenen Indexwerte deutlich kleiner als die Anzahl der verschiedenen *möglichen* Datensätze, d.h. werden sehr viele Datensätze gespeichert, dann können Kollisionen nicht vermieden werden (vgl. Taubenschlagprinzip oder Schubfachschluss).

Beispiel 53: *Sei nun* $N = 10$*, d.h. die gültigen Indexwerte sind* $0, \ldots, 9$*. Die zu speichernden Datensätze seien natürliche Zahlen und es soll* $h(x) = x \bmod 10$ *als Hashfunktion gewählt werden. Werden die Zahlen* 42*,* 60*,* 75 *und* 119 *in der Hashtable gespeichert, dann ergibt sich das folgende Bild:*

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Bucket	*60*		*42*			*75*				*119*

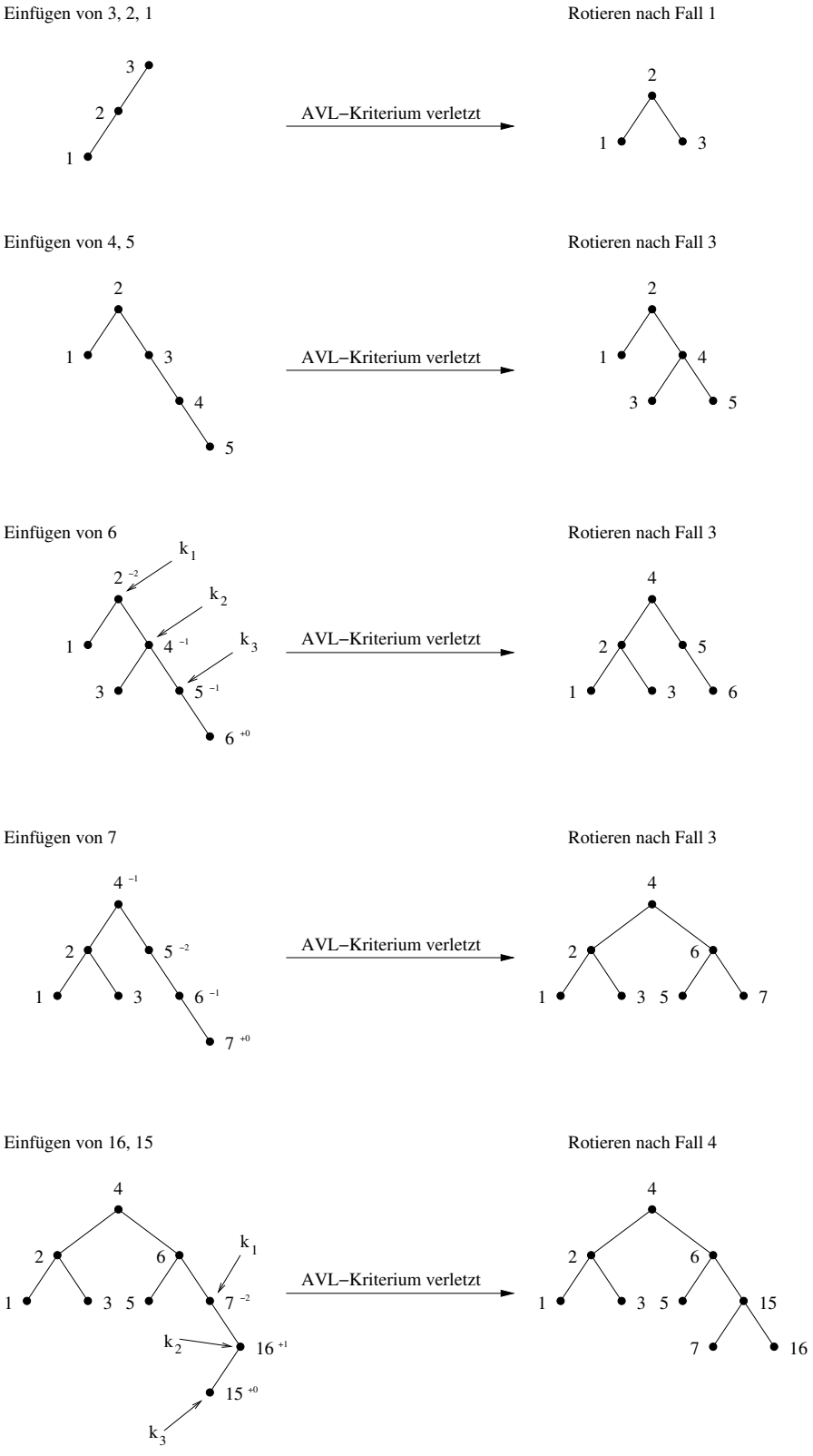


Abbildung 27: Einfügen in einen AVL-Baum

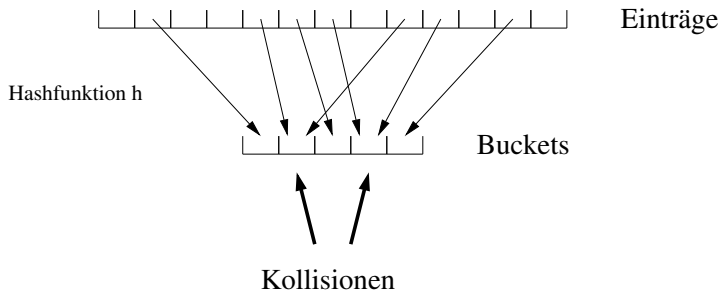


Abbildung 28: Hashing - Ein graphischer Überblick

Soll noch der Wert 69 *gespeichert werden, so kommt es zu einer Kollision.*

4.1. Hashfunktionen

Schon Beispiel 53 zeigt, dass die Wahl der Hashfunktion großen Einfluss auf die Qualität der Hashens hat. Weiterhin zeigt sich aber auch, dass eine konkrete Hashfunktion zu dem Datentyp der verwalteten Datensäte passen muss. Damit stellt sich die Frage, wie man in der Praxis brauchbare Hashfunktionen für Standarddatentypen wie `int`, `String` oder `double` konstruieren kann.

Integer: Oft wählt man $h(x) = x \bmod N$, wobei N eine „große“ Primzahl sein sollte, die nicht in der Nähe einer 2er Potenz liegt (also z.B. *nicht* $N = 509$ oder $N = 1021$).

Wird z.B. $N = 2^j$ gewählt, denn werden alle ungeraden Werte auf ungerade Indizes abgebildet, das zu einer schlechten Verteilung führt.

String: Sei c ein beliebiger Buchstabe, dann ist $\texttt{ASCII}(c)$ sei ASCII-Wert (z.B. $\texttt{ASCII}(A) = 65$). Sei s ein String der Länge n, dann berechnen wir den Hashwert durch

$$h(s) = \sum_{i=0}^{n-1} a_i \cdot \texttt{ASCII}(s[i]) \bmod N$$

Dabei werden die a_i's bei der Konstruktion von h zufällig gewählt und sind dann für diese Hashfunktion unveränderlich.

Fließkommazahlen: Addiere den Exponenten und die Mantisse um eine natürliche Zahl zu erhalten, die dann wie eine Integerzahl behandelt werden kann.

Beispiel 54: *Es sind Studentendatensätze zu hashen, die durch eine* 6*-stellige Matrikelnummer identifiziert werden. Matrikelnummern werden fortlaufend vergeben. Dann ist die Hashfunktion* $h(e)$ = *„die ersten zwei Ziffern der Matrikelnummer* e*“ sicherlich schlecht und führt zu sehr vielen Kollisionen.*

Es ist einsichtig, dass eine Hashfunktion schnell berechenbar sein muss. Insbesondere darf die Berechnung des Hashwerts nicht länger dauern als das eigentlich Einfügen der Daten.

4.2. Kollisionsbehandlung

Werden zwei Datensätze dem gleichen Bucket zugeteilt, dann spricht man von einer *Kollision.* Wir benötigen also Methoden um Kollisionen zu behandeln. Die führt zu folgender Idee:

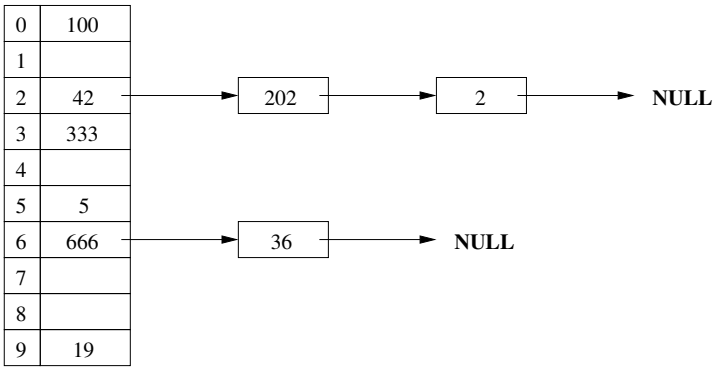


Abbildung 29: Beispiel für die Kollisionsbehandlung mit linearen Listen

- Jeder Bucket wird um eine Liste oder Suchbaum erweitert, in denen die kollidierenden Datensätze untergebracht werden.
- Im Fall einer Kollision such man nach einer alternativen Position. Diese Technik nennt man *Sondieren*.

4.2.1. Verkettung der kollidierenden Datensätze

Sei $N = 10$ und $h(x) = x \bmod 10$. Werden die Datensätze 100, 42, 5, 333, 19, 666, 202, 36 und 2 in eine Hashtabelle eingetragen, so ergibt sich Bild 29.

4.2.2. Sondieren

Lineares Sondieren Sei H das benutzte Feld und h die Hashfunktion. Ist das Element e zu speichern und ist $H[h(e)]$ schon belegt, dann suchen wir in der Folge

$$H[h(e)], H[h(e) + 1 \bmod N], H[h(e) + 2 \bmod N], H[h(e) + 3 \bmod N], \ldots$$

nach der ersten freien Position. Diese Technik ist als *lineares Sondieren* bekannt. Natürlich muss diese Veränderung auch beim Suchen berücksichtigt werden, da man ein Element nun nicht mehr unbedingt am Index $h(e)$ findet. Dieses Verfahren kann man auch leicht modifizieren, indem man für eine Konstante $c \in \mathbb{N}$ mit $\mathrm{ggT}(c, N) = 1$ in der Folge

$$H[h(e)], H[h(e) + 1 \cdot c \bmod N], H[h(e) + 2 \cdot c \bmod N], H[h(e) + 3 \cdot c \bmod N], \ldots$$

sucht. Lineares Sondieren neigt (insbesondere für $c = 1$) zur Bildung von „Klumpen“, wodurch sich die Laufzeit für Suchen und Einfügen erheblich verlängern kann.

Quadratisches Sondieren Das Verfahren der linearen Sondierung kann leicht durch das Suchen in der Folge

$$H[h(e)], H[h(e) + 1 \bmod N], H[h(e) + 4 \bmod N], H[h(e) + 9 \bmod N], \ldots, H[h(e) + i^2 \bmod N], \ldots$$

ersetzt werden. Diese Vorgehensweise einen freien Bucket zu finden ist als *quadratisches Sondieren* bekannt. Dabei wird eine übermäßige „Klumpenbildung“ vermieden, allerdings gehen dann auch Laufzeitvorteile aufgrund von Caching verloren (vgl. weit entfernte Datensätze sind nicht mehr im Cache enthalten und müssen zeitaufwändig nachgeladen werden).

Zu beachten ist, dass bei Sondierungsverfahren die entsprechenden Einträge nie gelöscht werden, sondern nur als gelöscht *markiert* werden dürfen, da sonst Lücken in der Sondierungsfolge auftreten können, die den Suchprozess unmöglich machen würden.

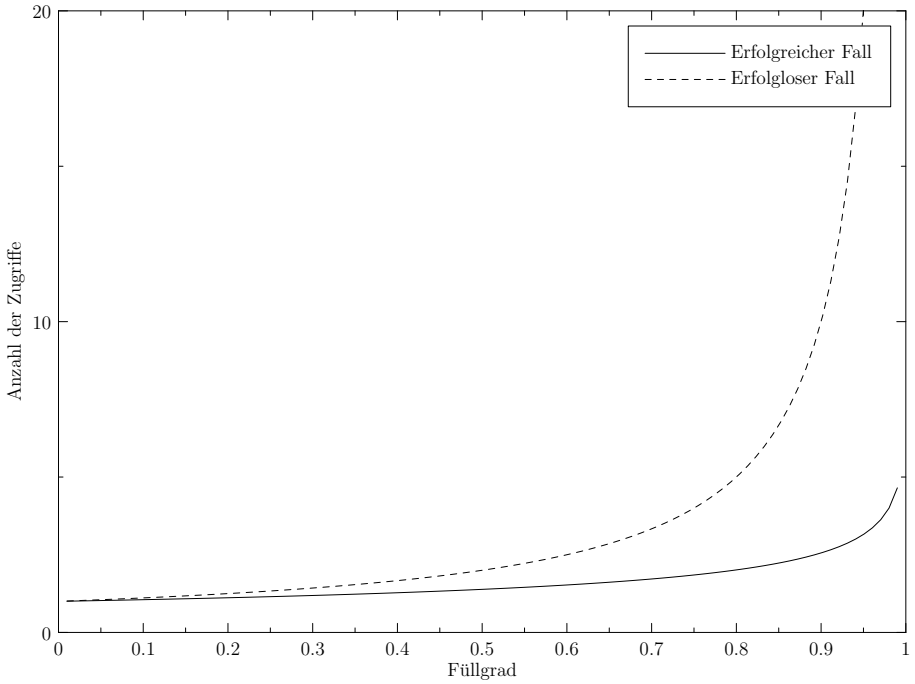


Abbildung 30: Vergleich des Aufwands für die erfolglose und die erfolgreiche Suche

4.3. Das Laufzeitverhalten von Hashingverfahren

Je höher der Füllgrad einer Hashtabelle ist, desto schlechter wird die Effizienz beim Suchen und Einfügen, weil die Wahrscheinlichkeit für Kollisionen steigt. Im schlimmsten Fall können dadurch $N - 1$ Kollisionen beim Sondieren auftreten (Hashtabelle vollständig besetzt). In diesem Fall würde die Verwendung von Suchbäumen, anstatt von linearen Listen wie im Abschnitt 4.2.1, zumindest einen Aufwand von $O(\log n)$ garantieren.

Verteilt die Hashfunktion die Datensätze „perfekt“, also völlig gleichmäßig und treten nur wenige Kollisionen auf, dann ergeben sich die folgenden Aufwände:

- Einfügen: $O(1)$
- Löschen: $O(1)$ (wenn Einträge nur als gelöscht markiert werden)

Natürlich können Kollisionen nicht vernachlässigt werden. Seien m Datensätze in einer Hashtabelle mit N Einträgen gespeichert, dann definieren wir den *Füllgrad* α durch

$$\alpha =_{\text{def}} \frac{m}{N}$$

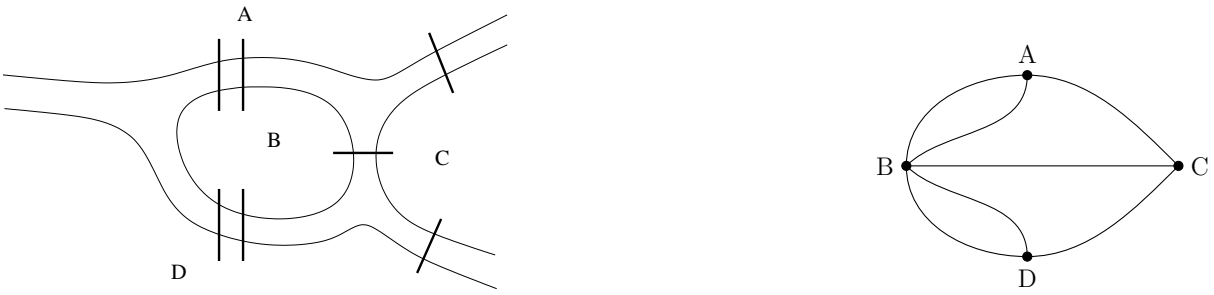
D.h. wir brauchen bei einer erfolglosen Suche eines freien Platzes

$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots = \frac{1}{1-\alpha} \text{ (geometrische Reihe)}$$

Zugriffe, wenn wir *näherungsweise* annehmen, dass wir mit einer Wahrscheinlichkeit von α einen zweiten, α^2 einen dritten Zugriff, usw. benötigen. Für die erfolgreiche Suche ergeben sich im Mittel

$$\frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$$

Zugriffe. Abbildung 30 veranschaulicht, dass sich der erfolglose und der erfolgreiche Fall für niedrige Füllgrade kaum unterscheidet, wogegen für hohe Füllgrade ein dramatischer Unterschied eintritt.



(a) grober Stadtplan (b) formalisierter Stadtplan

Abbildung 31: Das Königsberger-Brückenproblem

5. Graphen und Graphenalgorithmen

5.1. Einführung

Sehr viele Probleme lassen sich durch Objekte und Verbindungen oder Beziehungen zwischen diesen Objekten beschreiben. Ein schönes Beispiel hierfür ist das *Königsberger Brückenproblem*, das 1736 von Leonhard Euler[12] formuliert und gelöst wurde. Zu dieser Zeit hatte Königsberg[13] genau sieben Brücken, wie die sehr grobe Karte in Abbildung 31 zeigt.

Die verschiedenen Stadtteile sind dabei mit A-D bezeichnet. Euler stellte sich nun die Frage, ob es möglich ist, einen Spaziergang in einem beliebigen Stadtteil zu beginnen, jede Brücke *genau einmal* zu überqueren und den Spaziergang am Startpunkt zu beenden. Ein solcher Weg soll *Euler-Spaziergang* heißen. Die Frage lässt sich leicht beantworten, wenn der Stadtplan wie nebenstehend formalisiert wird.

Die Stadtteile sind bei der Formalisierung zu Knoten geworden und die Brücken werden durch Kanten zwischen den Knoten symbolisiert[14]. Angenommen es gäbe in Königsberg einen Euler-Spaziergang, dann müsste für jeden Knoten in Abbildung 31 die folgende Eigenschaft erfüllt sein: die Anzahl der Kanten die mit einem Knoten verbunden sind ist gerade, weil für jede Ankunft (über eine Brücke) in einem Stadtteil ein Verlassen eines Stadtteil (über eine Brücke) notwendig ist.

5.2. Grundlagen

Die Theorie der Graphen ist heute zu einem unverzichtbaren Bestandteil der Informatik geworden. Viele Probleme, wie z.B. das Verlegen von Leiterbahnen auf einer Platine, die Modellierung von Netzwerken oder die Lösung von Routingproblemen in Verkehrsnetzen benutzen Graphen oder Algorithmen, die Graphen als Datenstruktur verwenden. Auch schon bekannte Datenstrukturen wie Listen und Bäume können als Graphen aufgefasst werden. All dies gibt einen Anhaltspunkt, dass die Graphentheorie eine sehr zentrale Rolle für die Informatik spielt und vielfältige Anwendungen hat. In diesem Kontext ist es wichtig zu bemerken, dass der Begriff des Graphen in der Informatik *nicht* im Sinne von Graph einer Funktion gebraucht wird, sondern wie folgt definiert ist:

Definition 55: *Ein* gerichteter Graph $G = (V, E)$ *ist ein Paar, das aus einer Menge von* Knoten V *und einer Menge von* Kanten $E \subseteq V \times V$ *(*Kantenrelation*) besteht. Eine Kante* $k = (u, v)$ *aus* E *kann als Verbindung zwischen den Knoten* $u, v \in V$ *aufgefasst werden. Aus diesem Grund*

[12] Der Schweizer Mathematiker Leonhard Euler wurde 1707 in Basel geboren und starb 1783 in St. Petersburg.
[13] Königsberg heißt heute Kaliningrad.
[14] Abbildung 31 nennt man *Multigraph*, denn hier starten mehrere Kanten von *einem* Knoten und enden in *einem* anderen Knoten.

nennt man u auch Startknoten *und v* Endknoten. *Zwei Knoten, die durch eine Kante verbunden sind, heißen auch* benachbart *oder* adjazent.
Ein Graph $H = (V', E')$ mit $V' \subseteq V$ und $E' \subset E$ heißt Untergraph *von G.*

Ein Graph (V, E) heißt *endlich* gdw. die Menge der Knoten V endlich ist. Obwohl man natürlich auch unendliche Graphen betrachten kann, werden wir uns in diesem Abschnitt nur mit endlichen Graphen beschäftigen, da diese für den Informatiker von großem Nutzen sind.

Da wir eine Kante (u, v) als Verbindung zwischen den Knoten u und v interpretieren können, bietet es sich an, Graphen durch Diagramme darzustellen. Dabei wird die Kante (u, v) durch einen Pfeil von u nach v dargestellt. Drei Beispiele für eine bildliche Darstellung von gerichteten Graphen finden sich in Abbildung 32.

5.3. Einige Eigenschaften von Graphen

Der Graph in Abbildung 32(c) hat eine besondere Eigenschaft, denn offensichtlich kann man die Knotenmenge $V_{1c} = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ in zwei disjunkte Teilmengen $V_{1c}^l = \{0, 1, 2, 3\}$ und $V_{1c}^r = \{4, 5, 6, 7, 8\}$ so aufteilen, dass keine Kante zwischen zwei Knoten aus V_{1c}^l oder V_{1c}^r verläuft.

Definition 56: *Ein Graph $G = (V, E)$ heißt* bipartit*, wenn gilt:*

1. *Es gibt zwei Teilmengen V^l und V^r von V mit $V = V^l \cup V^r$, $V^l \cap V^r = \emptyset$ und*
2. *für jede Kante $(u, v) \in E$ gilt $u \in V^l$ und $v \in V^r$.*

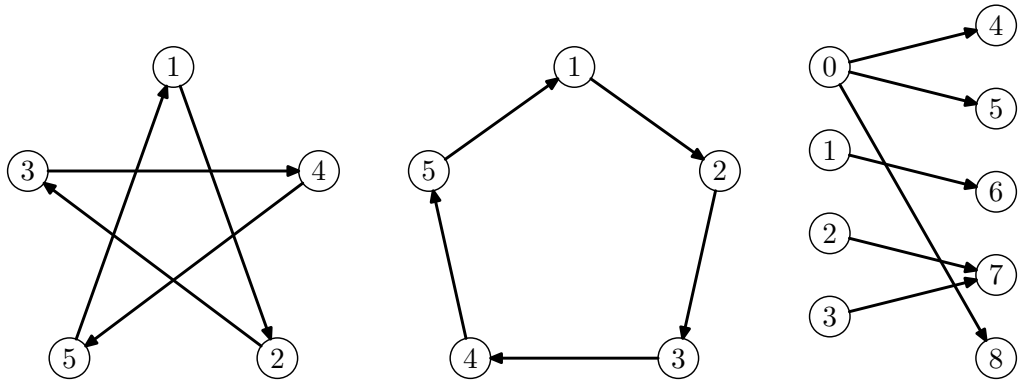
Bipartite Graphen haben viele Anwendungen, weil man jede binäre Relation $R \subseteq A \times B$ mit $A \cap B = \emptyset$ ganz natürlich als bipartiten Graph auffassen kann, dessen Kanten von Knoten aus A zu Knoten aus B laufen.

Beispiel 57: *Gegeben sei ein bipartiter Graph $G = (V, E)$ mit $V = V^F \cup V^M$ und $V^F \cap V^M = \emptyset$. Die Knoten aus V^F symbolisieren Frauen und V^M symbolisiert eine Menge von Männern. Kann sich eine Frau vorstellen einen Mann zu heiraten, so wird der entsprechende Knoten aus V^F mit dem passenden Knoten aus V^M durch eine Kante verbunden. Eine* Heirat *ist nun eine Kantenmenge $H \subseteq E$, so dass keine zwei Kanten aus H einen gemeinsamen Knoten besitzen. Das* Heiratsproblem *ist nun die Aufgabe für G eine Heirat H zu finden, so dass alle Frauen heiraten können, d.h. es ist das folgende Problem zu lösen:*

PROBLEM:	*MARRIAGE*
EINGABE:	*Bipartiter Graph $G = (V, E)$ mit $V = V^F \cup V^M$ und $V^F \cap V^M = \emptyset$*
AUSGABE:	*Eine Heirat H mit $\#H = \#V^F$*

Im Beispielgraphen 32(c) gibt es keine Lösung für das Heiratsproblem, denn für die Knoten ($\triangleq$ Kandidatinnen) 2 und 3 existieren nicht ausreichend viele Partner, d.h. keine Heirat in diesem Graphen enthält zwei Kanten die sowohl 2 als auch 3 als Startknoten haben.

Obwohl dieses Beispiel auf den ersten Blick nur von untergeordneter Bedeutung erscheint, kann man es auf eine Vielfalt von Anwendungen übertragen. Immer wenn die Elemente zweier disjunkter Mengen durch eine Beziehung verbunden sind, kann man dies als bipartiten Graphen auffassen. Sollen nun die Bedürfnisse der einen Menge völlig befriedigt werden, so ist dies wieder ein Heiratsproblem. Beispiele mit mehr praktischem Bezug finden sich u.a. bei Beziehungen zwischen Käufern und Anbietern.



(a) Ein gerichteter Graph mit 5 Knoten

(b) Ein planarer gerichteter Graph mit 5 Knoten

(c) Ein gerichteter bipartiter Graph

Abbildung 32: Beispiele für gerichtete Graphen

Oft beschränken wir uns auch auf eine Unterklasse von Graphen, bei denen die Kanten keine „Richtung“ haben (siehe Abbildung 33) und einfach durch eine Verbindungslinie symbolisiert werden können:

Definition 58: *Sei $G = (V, E)$ ein Graph. Ist die Kantenrelation E* symmetrisch*, d.h. gibt es zu jeder Kante $(u, v) \in E$ auch eine Kante $(v, u) \in E$ (siehe auch Abschnitt A.2.1), dann bezeichnen wir G als* ungerichteten Graphen *oder kurz als* Graph*.*

Es ist praktisch, die Kanten (u, v) und (v, u) eines ungerichteten Graphen als Menge $\{u, v\}$ mit zwei Elementen aufzufassen. Diese Vorgehensweise führt zu einem kleinen technischen Problem. Eine Kante (u, u) mit gleichem Start- und Endknoten nennen wir, entsprechend der intuitiven Darstellung eines Graphens als Diagramm, *Schleife*. Wandelt man nun solch eine Kante in eine Menge um, so würde nur eine einelementige Menge entstehen. Aus diesem Grund legen wir fest, dass ungerichtete Graphen *schleifenfrei* sind.

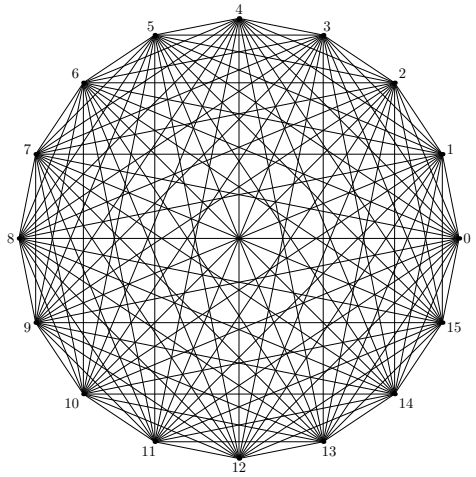
Definition 59: *Der (ungerichtete) Graph $K = (V, E)$ heißt* vollständig*, wenn für alle $u, v \in V$ mit $u \neq v$ auch $(u, v) \in E$ gilt, d.h. jeder Knoten des Graphen ist mit allen anderen Knoten verbunden. Ein Graph $O = (V, \emptyset)$ ohne Kanten wird als* Nullgraph *bezeichnet.*

Mit dieser Definition ergibt sich, dass die Graphen in Abbildung 33(a) und Abbildung 33(b) vollständig sind. Der Nullgraph $(V, \emptyset)$ ist Untergraph jedes beliebigen Graphen (V, E). Diese Definitionen lassen sich natürlich auch analog auf gerichtete Graphen übertragen.

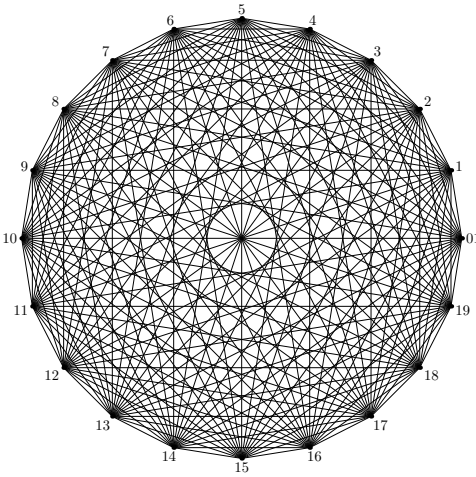
Definition 60: *Sei $G = (V, E)$ ein gerichteter Graph und $v \in V$ ein beliebiger Knoten. Der* Ausgrad *von v (kurz:* $\text{outdeg}(v)$*) ist dann die Anzahl der Kanten in G, die v als Startknoten haben. Analog ist der* Ingrad *von v (kurz:* $\text{indeg}(v)$*) die Anzahl der Kanten in G, die v als Endknoten haben.*

Bei ungerichteten Graphen gilt für jeden Knoten $\text{outdeg}(v) = \text{indeg}(v)$*. Aus diesem Grund schreiben wir kurz* $\deg(v)$ *und bezeichnen dies als* Grad von v*. Ein Graph G heißt* regulär *gdw. alle Knoten von G den gleichen Grad haben.*

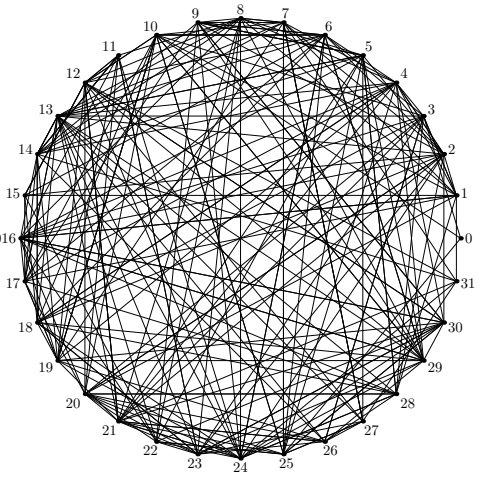
Die Diagramme der Graphen in den Abbildungen 32 und 33 haben die Eigenschaft, dass sich einige Kanten schneiden. Es stellt sich die Frage, ob man diese Diagramme auch so zeichnen kann, dass keine Überschneidungen auftreten. Diese Eigenschaft von Graphen wollen wir durch die folgende Definition festhalten:



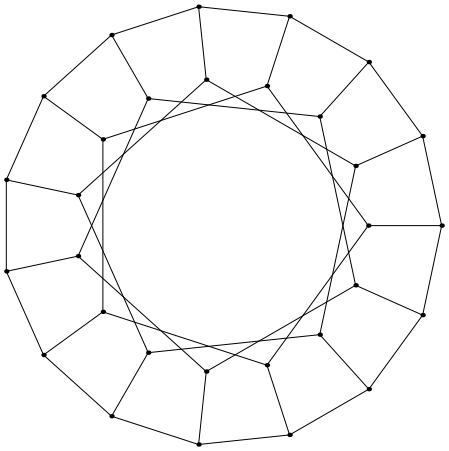
(a) Vollständiger ungerichteter Graph K_{16}



(b) Vollständiger ungerichteter Graph K_{20}



(c) Zufälliger Graph mit 32 Knoten



(d) Regulärer Graph mit Grad 3

Abbildung 33: Beispiele für ungerichtete Graphen

Definition 61: *Ein Graph G heißt* planar*, wenn sich sein Diagramm ohne Überschneidungen zeichnen läßt.*

Beispiel 62: *Der Graph in Abbildung 32(a) ist, wie man leicht nachprüfen kann, planar, da die Diagramme aus Abbildung 32(a) und 32(b) den gleichen Graphen repräsentieren.*

Auch planare Graphen haben eine anschauliche Bedeutung. Der Schaltplan einer elektronischen Schaltung kann als Graph aufgefasst werden. Die Knoten entsprechen den Stellen an denen die Bauteile aufgelötet werden müssen, und die Kanten entsprechen den Leiterbahnen auf der Platine. In diesem Zusammenhang bedeutet planar, ob man die Leiterbahnen kreuzungsfrei verlegen kann, d.h. ob es möglich ist, eine Platine zu fertigen, die mit einer Kupferschicht auskommt. In der Praxis kommen oft Platinen mit mehreren Schichten zum Einsatz („Multilayer-Platine"). Ein Grund dafür kann sein, dass der „Schaltungsgraph" nicht planar war und deshalb mehrere Schichten benötigt werden. Da Platinen mit mehreren Schichten in der Fertigung deutlich teurer sind als solche mit einer Schicht, hat die Planaritätseigenschaft von Graphen somit auch unmittelbare finanzielle Auswirkungen.

5.4. Wege, Kreise, Wälder und Bäume

Definition 63: *Sei $G = (V, E)$ ein Graph und $u, v \in V$. Eine Folge von Knoten $u_0, \ldots, u_l \in V$ mit $u = u_0$, $v = u_l$ und $(u_i, u_{i+1}) \in E$ für $0 \leq i \leq l-1$ heißt* Weg von u nach v der Länge l*. Der Knoten u wird* Startknoten *und v wird* Endknoten *des Wegs genannt.*

Ein Weg, bei dem Start- und Endknoten gleich sind, heißt geschlossener Weg*. Ein geschlossener Weg, bei dem kein Knoten außer dem Startknoten mehrfach enthalten ist, wird* Kreis *genannt.*

Mit Definition 63 wird klar, dass der Graph in Abbildung 32(a) den Kreis $1, 2, 3, \ldots, 5, 1$ mit Startknoten 1 hat.

Definition 64: *Sei $G = (V, E)$ ein Graph. Zwei Knoten $u, v \in V$ heißen* zusammenhängend*, wenn es einen Weg von u nach v gibt. Der Graph G heißt* zusammenhängend*, wenn jeder Knoten von G mit jedem anderen Knoten von G zusammenhängt.*

Sei G' ein zusammenhängender Untergraph von G mit einer besonderen Eigenschaft: Nimmt man einen weiteren Knoten von G zu G' hinzu, dann ist der neu entstandene Graph nicht mehr zusammenhängend, d.h. es gibt keinen Weg zu diesem neu hinzugekommenen Knoten. Solch einen Untergraph nennt man Zusammenhangskomponente*.*

Offensichtlich sind die Graphen in den Abbildungen 32(a), 33(a), 33(b) und 33(d) zusammenhängend und haben genau eine Zusammenhangskomponente. Man kann sich sogar leicht überlegen, dass die Eigenschaft „u hängt mit v" zusammen eine *Äquivalenzrelation* (siehe Abschnitt A.2.1) darstellt.

Mit Hilfe der Definition des geschlossenen Wegs lässt sich nun der Begriff der Bäume definieren, die eine sehr wichtige Unterklasse der Graphen darstellen.

Definition 65: *Ein Graph G heißt*

- Wald*, wenn es keinen geschlossenen Weg mit Länge ≥ 1 in G gibt und*
- Baum*, wenn G ein zusammenhängender Wald ist, d.h. wenn er nur genau eine Zusammenhangskomponente hat.*

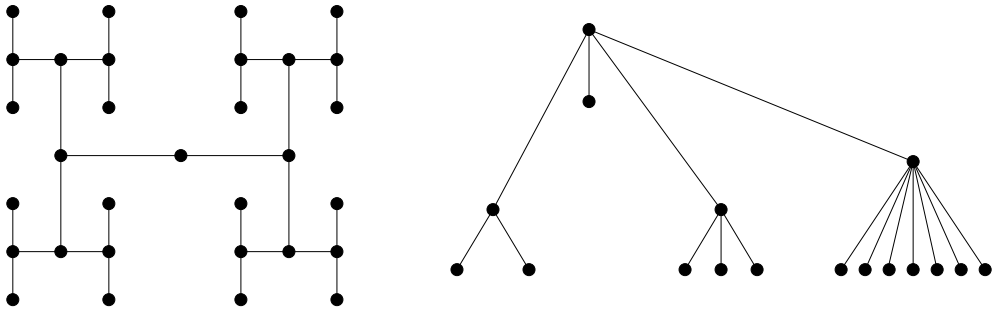


Abbildung 34: Ein Wald mit zwei Bäumen

5.5. Die Repräsentation von Graphen und einige Algorithmen

Nachdem Graphen eine große Bedeutung sowohl in der praktischen als auch in der theoretischen Informatik erlangt haben, stellt sich noch die Frage, wie man Graphen effizient als Datenstruktur in einem Computer ablegt. Dabei soll es möglich sein, Graphen effizient zu speichern und zu manipulieren.

Die erste Idee, Graphen als dynamische Datenstrukturen zu repräsentieren, scheitert an dem relativ ineffizienten Zugriff auf die Knoten und Kanten bei dieser Art der Darstellung. Sie ist nur von Vorteil, wenn ein Graph nur sehr wenige Kanten enthält. Die folgende Methode der Speicherung von Graphen hat sich als effizient erwiesen und ermöglicht auch die leichte Manipulation des Graphens:

Definition 66: *Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{v_1, \ldots, v_n\}$. Wir definieren eine $n \times n$ Matrix $A_G = (a_{i,j})_{1 \leq i,j, \leq n}$ durch*

$$a_{i,j} = \begin{cases} 1, & \textit{falls } (v_i, v_j) \in E \\ 0, & \textit{sonst} \end{cases}$$

Die so definierte Matrix A_G mit Einträgen aus der Menge $\{0, 1\}$ heißt Adjazenzmatrix *von G.*

Beispiel 67: *Für den gerichteten Graphen aus Abbildung 32(a) ergibt sich die folgende Adjazenzmatrix:*

$$A_{G_5} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Die Adjazenzmatrix eines ungerichteten Graphen erkennt man daran, dass sie spiegelsymmetrisch zur Diagonale von links oben nach rechts unten ist (die Kantenrelation ist symmetrisch) und dass die Diagonale aus 0-Einträgen besteht (der Graph hat keine Schleifen). Für den vollständigen

Graphen K_{16} aus Abbildung 33(a) ergibt sich offensichtlich die folgende Adjazenzmatrix:

$$A_{K_{16}} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Mit Hilfe der Adjazenzmatrix und Algorithmus 19 kann man leicht berechnen, ob ein Weg von einem Knoten u zu einem Knoten v existiert. Mit einer ganz ähnlichen Idee kann man auch leicht die Anzahl der Zusammenhangskomponenten berechnen (siehe Algorithmus 20). Dieser Algorithmus markiert die Knoten der einzelnen Zusammenhangskomponenten auch mit unterschiedlichen „Farben“, die hier durch Zahlen repräsentiert werden.

Algorithmus 19: Erreichbarkeit in Graphen

Eingabe: Ein Graph $G = (V, E)$ und zwei Knoten $u, v \in V$
Ergebnis: `true` wenn es einen Weg von u nach v gibt, `false` sonst

```
markiert = true;
markiere Startknoten u ∈ V;
while (markiert) do
    markiert = false;
    for (alle markierten Knoten w ∈ V) do
        if (w ∈ V ist adjazent zu einem unmarkierten Knoten w' ∈ V) then
            markiere Knoten w';
            markiert = true;
        end
    end
end
if (v ist markiert) then
    return true;
else
    return false;
end
```

Definition 68: *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Funktion der Form $f\colon V \to \{1, \ldots, k\}$ heißt* k-Färbung *des Graphen G. Anschaulich ordnet die Funktion f jedem Knoten eine von k verschiedenen Farben zu, die hier durch die Zahlen $1, \ldots, k$ symbolisiert werden. Eine*

Algorithmus 20: Zusammenhangskomponenten

Eingabe: Ein Graph $G = (V, E)$
Ergebnis: Anzahl der Zusammenhangskomponenten von G

```
kFarb = 0;
while (es gibt einen unmarkierten Knoten u ∈ V) do
    kFarb++;
    markiere u ∈ V mit kFarb;
    markiert=true;
    while (markiert) do
        markiert=false;
        for (alle mit kFarb markierten Knoten v ∈ V) do
            if (v ∈ V ist adjazent zu einem unmarkierten Knoten v' ∈ V) then
                markiere Knoten v' ∈ V mit kFarb;
                markiert=true;
            end
        end
    end
end
return kFarb;
```

Färbung heißt verträglich*, wenn für alle Kanten* $(u, v) \in E$ *gilt, dass* $f(u) \neq f(v)$*, d.h. zwei adjazente Knoten werden nie mit der gleichen Farbe markiert.*

Auch das Färbbarkeitsproblem spielt in der Praxis der Informatik eine wichtige Rolle. Ein Beispiel dafür ist die Planung eines Mobilfunknetzes. Dabei werden die Basisstationen eines Mobilfunknetzes als Knoten eines Graphen repräsentiert. Zwei Knoten werden mit einer Kante verbunden, wenn Sie geographisch so verteilt sind, dass sie sich beim Senden auf der gleichen Frequenz gegenseitig stören können. Existiert eine verträgliche k-Färbung für diesen Graphen, so ist es möglich, ein störungsfreies Mobilfunktnetz mit k verschiedenen Funkfrequenzen aufzubauen. Dabei entsprechen die Farben den verfügbaren Frequenzen. Bei der Planung eines solchen Mobilfunknetzes ist also das folgende Problem zu lösen:

Problem:	kCOL
Eingabe:	Ein ungerichteter Graph G und eine Zahl $k \in \mathbb{N}$.
Frage:	Gibt es eine verträgliche Färbung von G mit k Farben?

Dieses Problem gehört zu einer (sehr großen) Klasse von (praktisch relevanten) Problemen, für die bis heute keine effizienten Algorithmen bekannt sind (Stichwort: **NP**-Vollständigkeit). Vielfältige Ergebnisse der Theoretischen Informatik zeigen sogar, dass man nicht hoffen darf, dass ein schneller Algorithmus zur Lösung des Färbbarkeitsproblems existiert.

5.6. Einige Basisalgorithmen zur Traversierung von Graphen

5.6.1. Der Breitendurchlauf

Der *Breitendurchlauf* (engl. breath-first-search) ist der Ansatz die Knoten eines Graphen in Abhängigkeit von der Distanz zum Startknoten zu durchlaufen. Der folgende Algorithmus realisiert einen Breitendurchlauf mit Hilfe einer Warteschlange `Q`, berechnet für jeden Knoten den Abstand

`d` zum Startknoten und einen *Vorgänger* `p[u]` eines Knotens `u`. Da im Feld `p` zu jedem Knoten *genau ein* Vorgänger gespeichert ist, repräsentiert `p` einen Baum, der alle Knoten des Graphen umfasst. Ein solcher Baum wird als *aufspannender Baum* bezeichnet. Während des Durchlaufs werden die Knoten gefärbt. Dabei gilt weiß $\triangleq$ „unbesucht“, grau $\triangleq$ „besucht und Entfernung bestimmt“ und schwarz $\triangleq$ „vollständig bearbeitet“. Die Breitensuche wird durch Algorithmus 21 implementiert.

Beispiel 69: *Gegeben sei ein Graph G (siehe Abbildung 35) mit der Knotenmenge $V = \{r, s, t, u, v, w\}$. In Abbildung 35 sind alle Zwischenzustände angegeben, die entstehen, wenn man G mit Hilfe von Algorithmus 21 durchläuft.*

5.6.2. Der Tiefendurchlauf

Bei dem Breitendurchlauf werden die Knoten zuerst in der Breite, also im Abstand zum Startknoten, durchlaufen. Der *Tiefendurchlauf* untersucht einen gewählten Pfad immer erst so tief wie möglich (engl. depth-first-search). Wir realisieren den Tiefendurchlauf durch die Algorithmen 22 und 23. Ein Beispiel der Funktionsweise zeigt Abbildung 36.

5.7. Ausgewählte Graphenalgorithmen

5.7.1. Dijkstras Algorithmus

In vielen Anwendungen muss ein optimaler Weg zwischen einem Start- und einem Endknoten in einem gegebenen Graphen gefunden werden. Ein anschauliches Beispiel hierfür ist ein Routenplaner. Dazu ist eine Tabelle von Städten ($\triangleq$ Knoten eines Graphen) und Straßen ($\triangleq$ Kanten eines Graphen) gegeben. Zusätzlich sind die Kanten noch mit Entfernungen markiert. Mit Hilfe dieser Tabelle muss der Routenplaner eine möglichst kurze Verbindung zwischen Start und Ende finden. Natürlich lässt sich dieses Beispiel leicht abwandeln. Statt Entfernungen könnten als Kantenmarkierungen auch Fahrzeiten oder die Höhe der fälligen Mautgebühr verwendet werden. Im Falle von positiven Kantenmarkierungen kann das Problem eine optimale Route zu finden mit Hilfe von *Dijkstras Algorithmus* (vgl. Algorithmus 24) gelöst werden.

In Algorithmus 24 wird eine so genannte Prioritätswarteschlange verwendet. Diese speichert alle eingefügten Elemente immer *sortiert* und ermöglicht das Herauslesen des jeweils *kleinsten* Elements.

Wie schon erwähnt, kann Dijkstras Algorithmus nur den Fall von positiven Kantenmarkierungen lösen. Sollte also der Fall auftreten, dass man eine Erstattung von Mautgebühren ($\triangleq$ negative Gebühr) für die Benutzung einer Strecke bekommt, dann kann dieses verallgemeinerte Routenplanungsproblem nicht mehr mit diesem Algorithmus gelöst werden. In diesem Fall bietet sich der Bellman-Ford Algorithmus an. Da in dieser Algorithmus hier nicht mehr vorgestellt werden wird, soll auf [GK05] verwiesen werden.

5.7.2. Maximale Flüsse

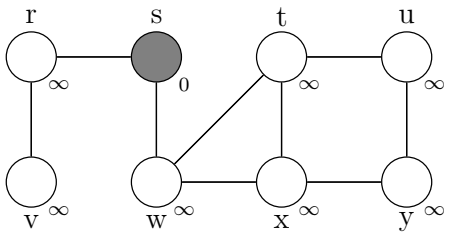
Gegeben sei ein gerichteter Graph $G = (V, E)$ mit gewichteten Kanten (siehe z.B. Abbildung 37), wobei diese Gewichte die maximale (Transport-) Kapazität der jeweiligen Kanten angeben, ein ausgezeichneter fester Startknoten s und ein Endknoten e.

Durch solch einem Graphen mit Kantengewichten ergibt sich unmittelbar die folgende Fragestellung:

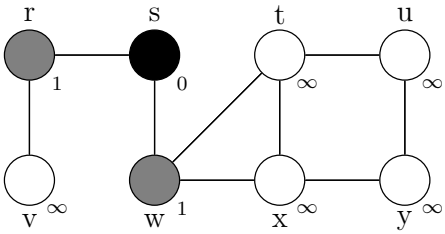
Algorithmus 21: Breitendurchlauf

Eingabe: Graph $G = (V, E)$, Startknoten s
Ergebnis: Alle Knoten werden gemäß Breitensuche durchlaufen

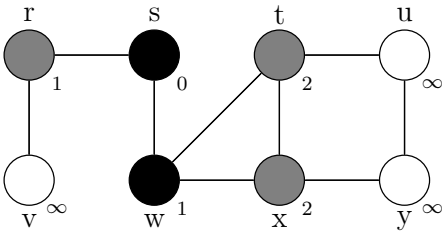
```
for (alle v ∈ V \ {s}) do
    /* Knoten v unbearbeitet */
    farbe[v] = weiss;
    /* Abstand undefiniert */
    d[v] = ∞;
    /* v hat kein Vorgänger */
    π[v] = null;
end
/* Informationen für den Startknoten eintragen */
farbe[s] = grau;
d[s] = 0;
π[s] = null;
erzeuge Warteschlange Q;
/* Startknoten in die Warteschlange eintragen */
Q.append(s);
while (!isEmpty(Q)) do
    /* Hole den nächsten Knoten der bearbeitet werden muss */
    u = Q.get();
    /* Alle direkten Nachfolger von u bearbeiten */
    for (alle Knoten v mit (u, v) ∈ E) do
        /* Knoten v schon besucht? */
        if (farbe[v] = weiss) then
            /* Informationen für den Knoten v eintragen */
            farbe[v] = grau;
            d[v] = d[u] + 1;
            π[v] = u;
            /* Knoten v muss später noch bearbeitet werden */
            Q.append(v);
        end
        /* Informationen für den Knoten v eintragen */
    end
    /* Knoten u wurde vollständig bearbeitet */
    farbe[u] = schwarz;
end
```



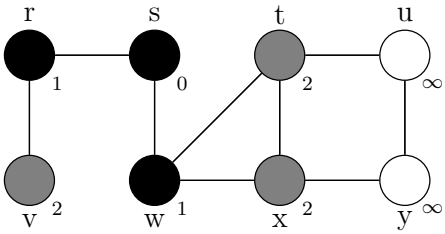
Start (Queue Q; s)



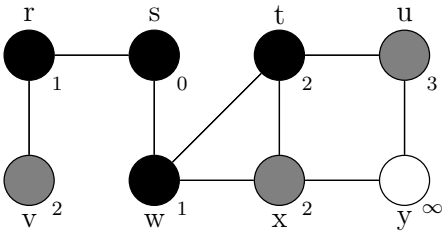
1. Durchlauf (Queue Q; w,r)



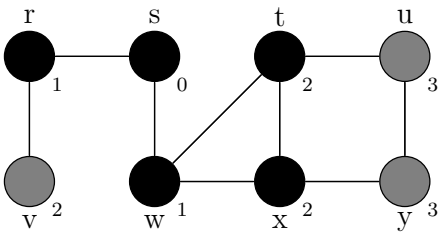
2. Durchlauf (Queue Q; r,t,x)



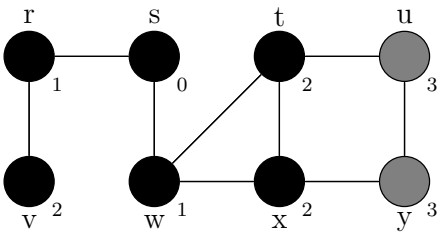
3. Durchlauf (Queue Q; t,x,v)



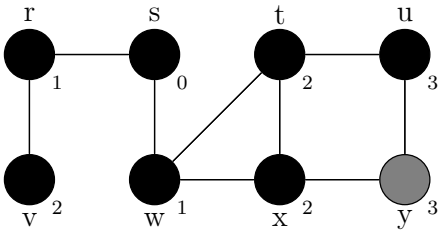
4. Durchlauf (Queue Q; x,v,u)



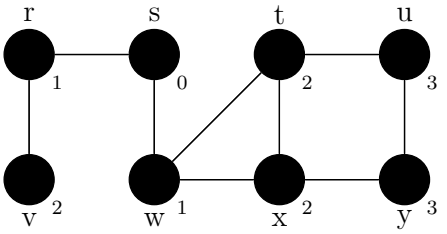
5. Durchlauf (Queue Q; v,u,y)



6. Durchlauf (Queue Q; u,y)



7. Durchlauf (Queue Q; y)



8. Durchlauf (Queue Q;)

Abbildung 35: Der Graph und alle Zwischenzustände aus Beispiel 69

Algorithmus 22: Tiefendurchlauf

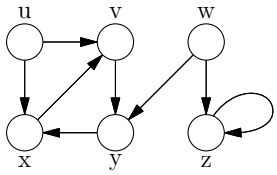
Eingabe: Graph $G = (V, E)$
Ergebnis: Alle Knoten werden gemäß Tiefensuche durchlaufen

```
for (alle v ∈ V) do
    /* Knoten v unbearbeitet */
    farbe[v] = weiss;
    /* v hat kein Vorgänger */
    π[v] = null;
end
/* Die Variable zeit ist global */
zeit = 0;
/* Alle Knoten ansehen */
for (alle u ∈ V) do
    /* Knoten u unbearbeitet? */
    if (farbe[u] == weiss) then
        /* Siehe Algorithmus 23 */
        DFS-Visit(G,u);
    end
end
```

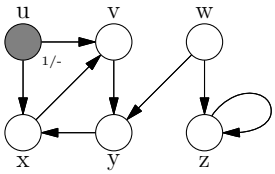
Algorithmus 23: DFS-Visit

Eingabe: Graph $G = (V, E)$, Startknoten u eines Pfads
Ergebnis: Alle Knoten werden gemäß Tiefensuche durchlaufen

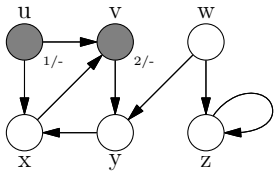
```
/* Knoten u ist nun besucht, Zeit erhöhen und Startzeit merken */
farbe[u] = grau;
zeit++;
d[u] = zeit;
/* Alle direkten Nachfolger von u bearbeiten */
for (alle Knoten v mit (u, v) ∈ E) do
    if (farbe[u] == weiss) then
        /* Vorgänger merken */
        π[v] = u;
        /* Rekursiver Aufruf */
        DFS − Visit(G, v);
    end
end
/* Knoten u ist nun vollständig bearbeitet, Endzeit merken */
farbe[u] = schwarz;
zeit++;
f[u] = zeit;
```



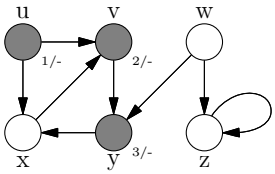
Start



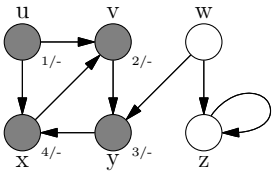
1. Durchlauf



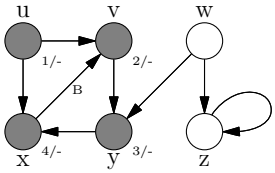
2. Durchlauf



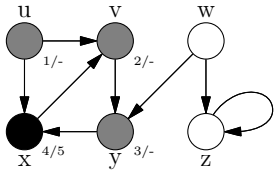
3. Durchlauf



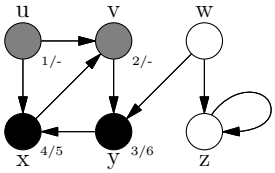
4. Durchlauf



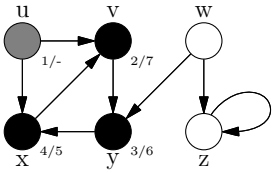
5. Durchlauf



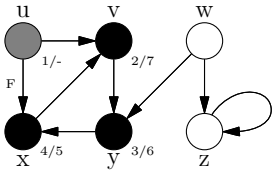
6. Durchlauf



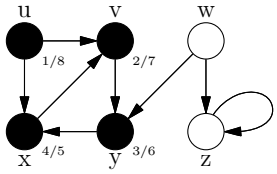
7. Durchlauf



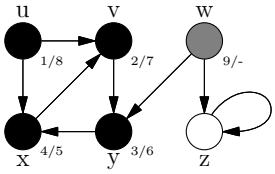
8. Durchlauf



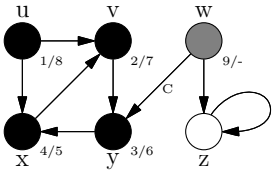
9. Durchlauf



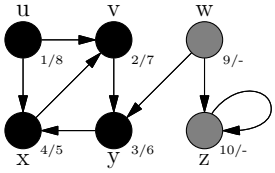
10. Durchlauf



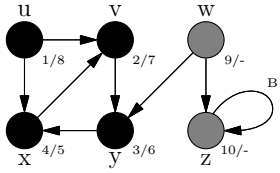
11. Durchlauf



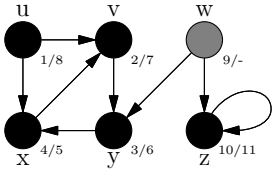
12. Durchlauf



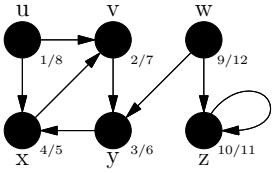
13. Durchlauf



14. Durchlauf



15. Durchlauf



16. Durchlauf

Abbildung 36: Ein Beispiel für die Tiefensuche

Algorithmus 24: Dijkstras Algorithmus

Eingabe: Graph $G = (V, E)$ mit Kantengewichten $\gamma((u, v))$ für eine Kante (u, v) und Startknoten s

Ergebnis: Feld $D[]$, das für jeden Knoten die (geringsten) Entfernung zum Startknoten enthält

```
begin
    /* Alle Knoten außer dem Startknoten abarbeiten */
    for (alle Knoten u ∈ V \ {s}) do
        /* Distanz von u auf den größten möglichen Wert setzen */
        D[u] = ∞;
    end
    /* Abstand des Startknoten zum Startknoten ist 0 */
    D[s] = 0;
    /* Leere Prioritätenwarteschlange erzeugen */
    PriorityQueue Q;
    /* Alle Paare von Knoten und Abstand in Q einfügen */
    for (alle Knoten u ∈ V) do
        Q.offer((D, D[u]));
    end
    /* noch nicht alle Knoten in der Warteschlange abgearbeitet? */
    while (!Q.isEmpty()) do
        /* Hole den Knoten u mit der kleinsten Distanz d */
        (u, d) = Q.poll();
        /* direkte Nachbarn, die noch bearbeitet werden müssen */
        for (für alle v ∈ V mit (u, v) ∈ E und für die v ∈ Q gilt) do
            /* Kürzeren Weg nach v gefunden? */
            if (D[u] + γ((u, v)) < D[v]) then
                /* Kürzere Distanz merken und Änderung in Q einbauen */
                D[v] = D[u] + γ((u, v));
                passe die neue Distanz des Knotens v in Q an;
            end
        end
    end
    /* Ergebnisarray zurückgeben */
    return D;
end
```

Was ist der maximale „Durchfluss“ von einem gegebenen Startknoten s zu einem gegebenen Endknoten e?

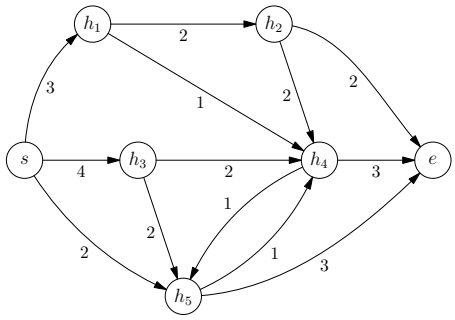


Abbildung 37: Beispiel für einen Graphen mit gewichteten Kanten

Anschaulich kann man dieses Problem z.B. als Netzwerk von Computern bzw. Routern auffassen, wobei die Kanten des Graphen die Datenleitungen darstellen und die Kantengewichte entsprechen z.B. der jeweiligen Übertragungskapazität. Die Knoten des Graphs stehen dann für den Computer bzw. Router selbst. Das Problem ist nun, die maximale Datenübertragungskapazität vom Start- zum Endknoten zu berechnen.

Definition 70: *Ein* Fluss F *legt für jede Kante einen* aktuellen Fluss f *fest, der unter der* maximalen Kapazität c *liegt. Die* Restkapazität *beträgt* $c - |f|$.

Definition 71: *Ein Fluss ist* korrekt*, wenn die folgenden Bedingungen für eine Kante* (u, v) *gelten:*

i) $|f((u,v))| \leq c \cdot ((u,v))$ *(„Die maximale Kapazität einer Kante wird nicht überschritten“)*

ii) $f((u,v)) = -f((v,u))$ *(„Konsistenz - bei gegenläufigen Kanten, die zwei Knoten verbinden, wird nur eine Richtung benutzt“)*

iii) Für alle Knoten $v \in V \setminus \{s, e\}$ *gilt*

$$\sum_{\substack{u \in V \\ (u,v) \in E}} f((u,v)) - \sum_{\substack{u \in V \\ (v,u) \in E}} f((u,v)) = 0$$

(„Bewahrung der Flüsse - in jeden Knoten, außer Start- und Endknoten, fließt genau soviel hinein wie heraus.“)

Für einen Graphen G mit Fluss F und einem Startknoten legen wir den aus dem Startknoten *abgehenden Fluss* wie folgt fest:

$$\mathrm{val}(G, F, s) =_{\mathrm{def}} \sum_{\substack{u \in V \\ (s,u) \in E}} f((s,v))$$

Dieser vom Startknoten abgehende Fluss muss natürlich mit dem in den Endknoten ankommenden Fluss übereinstimmen. Gesucht ist damit also das folgende Maximum

$$\max\{\mathrm{val}(G, F, s) \mid F \text{ ist ein korrekter Fluss, der bei } s \text{ startet und bei } e \text{ endet}\}.$$

Der Ford-Fulkerson Algorithmus Als *nutzbarer Pfad* wird ein (zyklenfreier[15]) Pfad vom Startknoten s zum Endknoten e bezeichnet, wenn er an allen Kanten eine verfügbare Kapazität $c - |f| \geq 1$ hat. Die Kapazität dieses Pfads ist dann das *Minimum* der einzelnen Kantenkapazitäten. Der Ford-Fulkerson Algorithmus (vgl. Algorithmus 25) berechnet dann den maximalen Fluss durch einen gegebenen gewichteten Graphen. Der Pseudocode des Ford-Fulkerson Algorithmus enthält keine explizite Angabe, wie nutzbare Pfade gefunden werden können, denn dies kann relativ leicht mit Hilfe eines Tiefendurchlaufs bewerkstelligt werden. Ein Beispiel zeigt Abbildung 38.

Algorithmus 25: Der Ford-Fulkerson Algorithmus

Eingabe: Graph $G = (V, E)$ mit Kantengewichten $\gamma((u, v))$ für eine Kante (u, v), Startknoten s und Endknoten e

Ergebnis: Ein markierter Graph G, sodass die vom Startknoten abgehenden Flüsse maximal sind.

```
begin
    markiere und initialisiere den Graph G mit dem leeren Fluss F_0;
    repeat
        wähle einen neuen nutzbaren Pfad p aus;
        füge den Fluss des Pfads p zum aktuellen Gesamtfluss hinzu und markiere den
         Graph entsprechend;
    until (es ist noch ein nutzbarer Pfad vorhanden);
    return G;
end
```

6. Komplexität

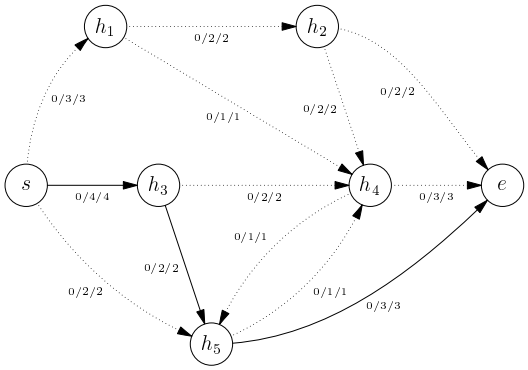
Für viele ständig auftretende Berechnungsprobleme, wie Sortieren, die arithmetischen Operationen (Addition, Multiplikation, Division), Fourier-Transformation etc., sind sehr effiziente Algorithmen konstruiert worden. Für wenige andere praktische Probleme weiß man, dass sie nicht oder nicht effizient algorithmisch lösbar sind. Im Gegensatz dazu ist für einen sehr großen Teil von Fragestellungen aus den verschiedensten Anwendungsbereichen wie Operations Research, Netzwerkdesign, Programmoptimierung, Datenbanken, Betriebssystem-Entwicklung und vielen mehr jedoch nicht bekannt, ob sie effiziente Algorithmen besitzen (vgl. die Abbildungen 43 und 44). Diese Problematik hängt mit der seit über vierzig Jahren[16] offenen $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage zusammen, wahrscheinlich gegenwärtig das wichtigste ungelöste Problem der theoretischen Informatik. Es wurde sogar vor einiger Zeit auf Platz 1 der Liste der so genannten *Millennium Prize Problems* des Clay Mathematics Institute gesetzt[17]. Diese Liste umfasst sieben offene Probleme aus der gesamten Mathematik. Das Clay Institute zahlt jedem, der eines dieser Probleme löst, eine Million US-Dollar.

In diesem Abschnitt werden die wesentlichen Begriffe aus dem Kontext des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems

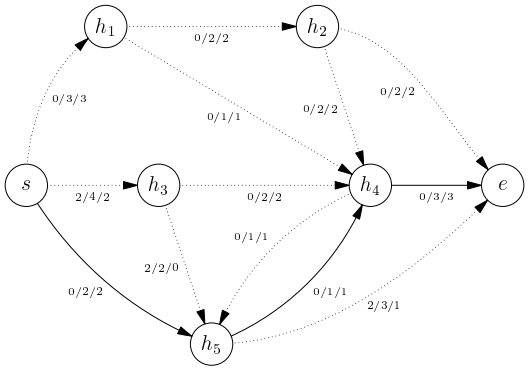
[15] Ein Pfad ist zyklenfrei genau dann, wenn kein Knoten mehr als einmal in diesem Pfad enthalten ist.

[16] Interessanterweise reicht die Geschichte des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems viel weiter in die Geschichte zurück. So fragte 1956 Kurt Gödel in einem Brief an John von Neumann schon „Given a first-order formula F and a natural number n, determine if there is a proof of F of length n“. Gödel war an der Anzahl von Schritten interessiert, die eine Turing-Maschine benötigt, um dieses Problem zu lösen. Dazu fragte er, ob dies in linearer oder quadratischer Zeit möglich ist, was man als eine Frühform der $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage auffassen könnte.

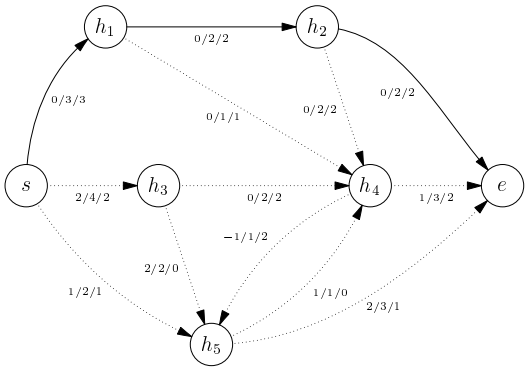
[17] siehe `http://www.claymath.org/millennium-problems`



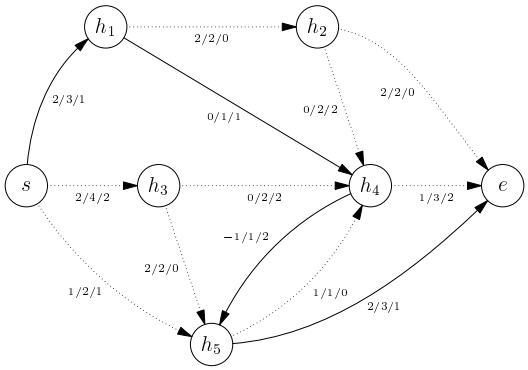
Auswahl des 1. Pfades



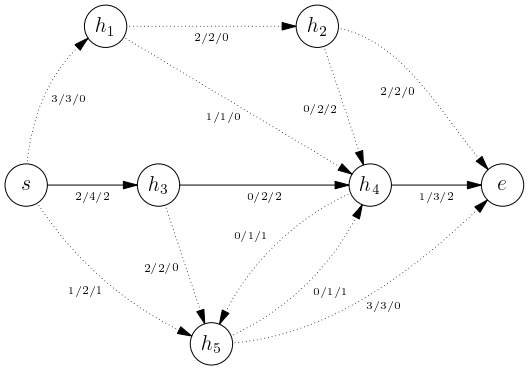
Auswahl des 2. Pfades



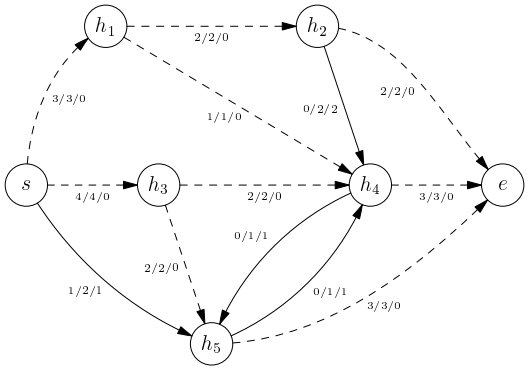
Auswahl des 3. Pfades



Auswahl des 4. Pfades



Auswahl des 5. Pfades



Ende der Berechnung

Abbildung 38: Ein Beispiel für den Ford-Fulkerson Algorithmus

und des Begriffes der **NP**-Vollständigkeit erläutert, um so die Grundlagen für das Verständnis derartiger Probleme zu schaffen und deren Beurteilung zu ermöglichen.

6.1. Effizient lösbare Probleme: die Klasse P

Jeder, der schon einmal mit der Aufgabe konfrontiert wurde, einen Algorithmus für ein gegebenes Problem zu entwickeln, kennt die Hauptschwierigkeit dabei: Wie kann ein effizienter Algorithmus gefunden werden, der das Problem mit möglichst wenigen Rechenschritten löst? Um diese Frage beantworten zu können, muss man sich zunächst einige Gedanken über die verwendeten Begriffe, nämlich „Problem“, „Algorithmus“, „Zeitbedarf“ und „effizient“, machen.

Was ist ein „Problem“? Jedem Programmierer ist diese Frage intuitiv klar: Man bekommt geeignete Eingaben, und das Programm soll die gewünschten Ausgaben ermitteln. Ein einfaches Beispiel ist das Problem MULT. (Jedes Problem soll mit einem eindeutigen Namen versehen und dieser in Großbuchstaben geschrieben werden.) Hier bekommt man zwei ganze Zahlen als Eingabe und soll das Produkt beider Zahlen berechnen, d.h. das Programm berechnet einfach eine zweistellige Funktion. Es hat sich gezeigt, dass man sich bei der Untersuchung von Effizienzfragen auf eine abgeschwächte Form von Problemen beschränken kann, nämlich sogenannte *Entscheidungsprobleme*. Hier ist die Aufgabe, eine gewünschte Eigenschaft der Eingaben zu testen. Hat die aktuelle Eingabe die gewünschte Eigenschaft, dann gibt man den Wert 1 ($\triangleq$ `true`) zurück (man spricht dann auch von einer *positiven Instanz* des Problems), hat die Eingabe die Eigenschaft nicht, dann gibt man den Wert 0 ($\triangleq$ `false`) zurück. Oder anders formuliert: Das Programm berechnet eine Funktion, die den Wert 0 oder 1 zurück gibt und partitioniert damit die Menge der möglichen Eingaben in zwei Teile: die Menge der Eingaben mit der gewünschten Eigenschaft und die Menge der Eingaben, die die gewünschte Eigenschaft nicht besitzen. Folgendes Beispiel soll das Konzept verdeutlichen:

PROBLEM: PARITY
EINGABE: Positive Integerzahl x
AUSGABE: Ist die Anzahl der Ziffern 1 in der Binärdarstellung von x ungerade?

Es soll also ein Programm entwickelt werden, das die Parität einer Integerzahl x berechnet. Eine mögliche Entscheidungsproblem-Variante des Problems MULT ist die folgende:

PROBLEM: MULT_D
EINGABE: Integerzahlen x, y, positive Integerzahl i
AUSGABE: Ist das i-te Bit in $x \cdot y$ gleich 1?

Offensichtlich sind die Probleme MULT und MULT_D gleich schwierig (oder leicht) zu lösen.

Im Weiteren wollen wir uns hauptsächlich mit Problemen beschäftigen, die aus dem Gebiet der Graphentheorie stammen. Das hat zwei Gründe. Zum einen können, wie sich noch zeigen wird, viele praktisch relevante Probleme mit Hilfe von Graphen modelliert werden, und zum anderen sind sie anschaulich und oft relativ leicht zu verstehen. Ein (ungerichteter) Graph G besteht aus einer Menge von Knoten V und einer Menge von Kanten E, die diese Knoten verbinden. Man schreibt: $G = (V, E)$. Ein wohlbekanntes Beispiel ist der Nikolausgraph: $G_N = \big(\{1,2,3,4,5\}, \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}\big)$. Es gibt also fünf Knoten $V = \{1,2,3,4,5\}$, die durch die Kanten in $E = \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}$ verbunden werden (siehe Abbildung 39).

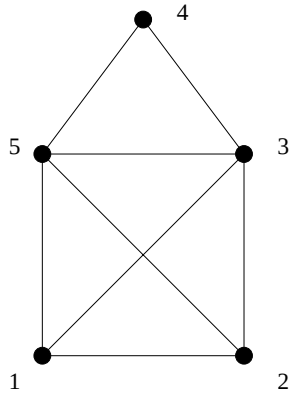


Abbildung 39: Der Graph G_N

Ein prominentes Problem in der Graphentheorie ist es, eine
sogenannte Knotenfärbung zu finden. Dabei wird jedem Knoten eine Farbe zugeordnet, und man verbietet, dass zwei Knoten, die durch eine Kante verbunden sind, die gleiche Farbe zugeordnet wird. Natürlich ist die Anzahl der Farben, die verwendet werden dürfen, durch eine feste natürliche Zahl k beschränkt. Genau wird das Problem, ob ein Graph k-färbbar ist, wie folgt beschrieben:

PROBLEM:	kCOL
EINGABE:	Ein Graph $G = (V, E)$
AUSGABE:	Hat G eine Knotenfärbung mit höchstens k Farben?

Offensichtlich ist der Beispielgraph G_N nicht mit drei Farben färbbar (aber mit 4 Farben, wie man leicht ausprobieren kann), und jedes Programm für das Problem 3COL müsste ermitteln, dass G_N die gewünschte Eigenschaft (3-Färbbarkeit) nicht hat.

Man kann sich natürlich fragen, was das künstlich erscheinende Problem 3COL mit der Praxis zu tun hat. Das folgende einfache Beispiel soll das verdeutlichen. Man nehme das Szenario an, dass ein großer Telefonprovider in einer Ausschreibung drei Funkfrequenzen für einen neuen Mobilfunkstandard erworben hat. Da er schon über ein Mobilfunknetz verfügt, sind die Sendemasten schon gebaut. Aus technischen Gründen dürfen Sendemasten, die zu eng stehen, nicht mit der gleichen Frequenz funken, da sie sich sonst stören würden. In der graphentheoretischen Welt modelliert man die Sendestationen mit Knoten eines Graphen, und „nahe“ zusammenstehende Sendestationen symbolisiert man mit einer Kante zwischen den Knoten, für die sie stehen. Die Aufgabe des Mobilfunkplaners ist es nun, eine 3-Färbung für den entstehenden Graphen zu finden. Offensichtlich kann das Problem verallgemeinert werden, wenn man sich nicht auf drei Farben/Frequenzen festlegt; dann aber ergeben sich genau die oben definierten Probleme kCOL für beliebige Zahlen k.

Als nächstes ist zu klären, was unter einem „Algorithmus“ zu verstehen ist. Ein Algorithmus ist eine endliche, formale Beschreibung einer Methode, die ein Problem löst (z.B. ein Programm in einer beliebigen Programmiersprache). Diese Methode muss also alle Eingaben mit der gesuchten Eigenschaft von den Eingaben, die diese Eigenschaft nicht haben, unterscheiden können. Man legt fest, dass der Algorithmus für erstere den Wert 1 und für letztere den Wert 0 ausgeben soll. Wie soll die „Laufzeit“ eines Algorithmus gemessen werden? Um dies festlegen zu können, muss man sich zunächst auf ein sogenanntes *Berechnungsmodell* festlegen. Das kann man damit vergleichen, welche Hardware für die Implementation des Algorithmus verwendet werden soll. Für die weiteren Analysen soll das folgende einfache C-artige Modell verwendet werden: Es wird (grob!) die Syntax von C verwendet und festgelegt, dass jede Anweisung in einem Schritt abgearbeitet werden kann. Gleichzeitig beschränkt man sich auf zwei Datentypen: einen Integer-Typ und zusätzlich Arrays dieses Integer-Typs (wobei Array-Grenzen nicht deklariert werden müssen, sondern sich aus dem Gebrauch ergeben). Dieses primitive Maschinenmodell ist deshalb geeignet, weil man zeigen kann, dass jeder so formulierte Algorithmus auf realen Computern implementiert werden kann, ohne eine substantielle Verlangsamung zu erfahren. (Dies gilt zumindest, wenn die verwendeten Zahlen nicht übermäßig wachsen, d.h., wenn alle verwendeten Variablen nicht zu viel Speicher belegen. Genaueres zu dieser Problematik – man spricht von der Unterscheidung zwischen *uniformem Komplexitätsmaß* und *Bitkomplexität* – findet sich in [Sch01, S. 62f].)

Umgekehrt kann man ebenfalls sagen, dass dieses einfache Modell die Realität genau genug widerspiegelt, da auch reale Programme ohne allzu großen Zeitverlust auf diesem Berechnungsmodell simuliert werden können. Offensichtlich ist die *Eingabe* der Parameter, von dem die Rechenzeit für einen festen Algorithmus abhängt. In den vergangenen Jahrzehnten, in denen das Gebiet der Analyse von Algorithmen entstand, hat die Erfahrung gezeigt, dass die Länge

der Eingabe, also die Anzahl der Bits, die benötigt werden um die Eingabe zu speichern, ein geeignetes und robustes Maß ist, in der die Rechenzeit gemessen werden kann. Auch der Aufwand, die Eingabe selbst festzulegen (zu konstruieren), hängt schließlich von ihrer Länge ab, nicht davon, ob sich irgendwo in der Eingabe eine 0 oder 1 befindet.

6.1.1. Das Problem der 2-Färbbarkeit

Das Problem der 2-Färbbarkeit ist wie folgt definiert:

PROBLEM: 2COL
EINGABE: Ein Graph $G = (V, E)$
AUSGABE: Hat G eine Knotenfärbung mit höchstens 2 Farben?

Es ist bekannt, dass dieses Problem mit einem sogenannten *Greedy-Algorithmus* gelöst werden kann: Beginne mit einem beliebigen Knoten in G (z.B. v_1) und färbe ihn mit Farbe 1. Färbe dann die Nachbarn dieses Knoten mit 2, die Nachbarn dieser Nachbarn wieder mit 1, usw. Falls G aus mehreren Komponenten (d.h. zusammenhängenden Teilgraphen) besteht, muss dieses Verfahren für jede Komponente wiederholt werden. G ist schließlich 2-färbbar, wenn bei obiger Prozedur keine inkorrekte Färbung entsteht. Diese Idee führt zu Algorithmus 26.

Die Laufzeit von Algorithmus 26 kann wie folgt abgeschätzt werden: Die erste **for**-Schleife benötigt n Schritte. In der **while**-Schleife wird entweder mindestens ein Knoten gefärbt und die Schleife dann erneut ausgeführt, oder es wird kein Knoten gefärbt und die Schleife dann verlassen; also wird diese Schleife höchstens n-mal ausgeführt. Innerhalb der **while**-Schleife finden sich drei ineinander verschachtelte **for**-Schleifen, die alle jeweils n-mal durchlaufen werden, und eine **while**-Schleife, die maximal n-mal durchlaufen wird.

Damit ergibt sich also eine Gesamtlaufzeit der Größenordnung n^4, wobei n die Anzahl der Knoten des Eingabe-Graphen G ist. Wie groß ist nun die Eingabelänge, also die Anzahl der benötigten Bits zur Speicherung von G? Sicherlich muss jeder Knoten in dieser Speicherung vertreten sein, d.h. also, dass mindestens n Bits zur Speicherung von G benötigt werden. Die Eingabelänge ist also mindestens n. Daraus folgt, dass die Laufzeit des Algorithmus also höchstens von der Größenordnung N^4 ist, wenn N die Eingabelänge bezeichnet.

Tatsächlich sind (bei Verwendung geeigneter Datenstrukturen wie Listen oder Queues) wesentlich effizientere Verfahren für 2COL möglich. Aber auch schon das obige einfache Verfahren zeigt: 2COL hat einen Polynomialzeitalgorithmus, also 2COL $\in$ **P**.

Alle Probleme für die Algorithmen existieren, die eine Anzahl von Rechenschritten benötigen, die durch ein beliebiges Polynom beschränkt ist, bezeichnet man mit **P** („**P**“ steht dabei für „**P**olynomialzeit“). Auch dabei wird die Rechenzeit in der Länge der Eingabe gemessen, d.h. in der Anzahl der Bits, die benötigt werden, um die Eingabe zu speichern (zu kodieren). Die Klasse **P** wird auch als Klasse der *effizient lösbaren Probleme* bezeichnet. Dies ist natürlich wieder eine idealisierte Auffassung: Einen Algorithmus mit einer Laufzeit n^{57}, wobei n die Länge der Eingabe bezeichnet, kann man schwer als effizient bezeichnen. Allerdings hat es sich in der Praxis gezeigt, dass für fast alle bekannten Probleme in **P** auch Algorithmen existieren, deren Laufzeit durch ein Polynom kleinen[18] Grades beschränkt ist.

In diesem Licht ist die Definition der Klasse **P** auch für praktische Belange von Relevanz. Dass eine polynomielle Laufzeit etwas substanziell Besseres darstellt als exponentielle Laufzeit (hier beträgt die benötigte Rechenzeit $2^{c \cdot n}$ für eine Konstante c, wobei n wieder die Länge der Eingabe bezeichnet), zeigt die Tabelle „Rechenzeitbedarf von Algorithmen“. Zu beachten ist, dass bei einem Exponentialzeit-Algorithmus mit $c = 1$ eine Verdoppelung der „Geschwindigkeit“

[18] Aktuell *scheint* es kein praktisch relevantes Problem aus **P** zu geben, für das es keinen Algorithmus mit einer Laufzeit von weniger als n^{12} gibt.

Algorithmus 26: Algorithmus zur Berechnung einer 2-Färbung eines Graphen

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$;
Ergebnis: 1 wenn es eine 2-Färbung für G gibt, 0 sonst

```
begin
    for (i = 1 to n) do
        Farbe[i] = 0;
    end
    Farbe[1] = 1;
    repeat
        aktKompoBearbeiten = false;
        for (i = 1 to n) do
            for (j = 1 to n) do
                                          /* Kante mit noch ungefärbten Knoten? */
                if (((v_i, v_j) ∈ E) und (Farbe[i] ≠ 0) und (Farbe[j] = 0)) then
                                          /* v_j bekommt eine andere Farbe als v_i */
                    Farbe[j] = 3 − Farbe[i];
                    aktKompoBearbeiten = true;
                                        /* Alle direkten Nachbarn von v_j prüfen */
                    for (k = 1 to n) do
                                          /* Kollision beim Färben aufgetreten? */
                        if (((v_j, v_k) ∈ E) und (Farbe[i] = Farbe[k])) then
                                            /* Kollision! Graph nicht 2-färbbar */
                            return 0;
                        end
                    end
                end
            end
        end
                /* Ist die aktuelle Zusammenhangskomponente völlig gefärbt? */
        if (not(aktKompoBearbeiten)) then
            i = 1;
                 /* Suche nach einer weiteren Zusammenhangskomponente von G */
            repeat
                  /* Liegt v_i in einer neuen Zusammenhangskomponente von G? */
                if (Farbe[i] = 0) then
                    Farbe[i] = 1;
                                    /* Neue Zusammenhangskomponente bearbeiten */
                    aktKompoBearbeiten = true;
                        /* Suche nach neuer Zusammenhangskomponente abbrechen */
                    weiterSuchen = false;
                end
                i = i + 1;
            until (not(weiterSuchen) und (i ≤ n));
        end
    until (aktKompoBearbeiten);
                                                        /* 2-Färbung gefunden */
    return 1.
end
```

Anzahl der	Eingabelänge n					
Takte	10	20	30	40	50	60
n	0,00001 Sekunden	0,00002 Sekunden	0,00003 Sekunden	0,00004 Sekunden	0,00005 Sekunden	0,00006 Sekunden
n^2	0,0001 Sekunden	0,0004 Sekunden	0,0009 Sekunden	0,0016 Sekunden	0,0025 Sekunden	0,0036 Sekunden
n^3	0,001 Sekunden	0,008 Sekunden	0,027 Sekunden	0,064 Sekunden	0,125 Sekunden	0,216 Sekunden
n^5	0,1 Sekunden	3,2 Sekunden	24,3 Sekunden	1,7 Minuten	5,2 Minuten	13,0 Minuten
2^n	0,001 Sekunden	1 Sekunde	17,9 Minuten	12,7 Tage	35,7 Jahre	366 Jahrhunderte
3^n	0,059 Sekunden	58 Minuten	6,5 Jahre	3855 Jahrhunderte	$2 \cdot 10^8$ Jahrhunderte	$1,3 \cdot 10^{13}$ Jahrhunderte

Abbildung 40: Rechenzeitbedarf von Algorithmen auf einem „1-MIPS“-Rechner

der verwendeten Maschine (also Taktzahl pro Sekunde) es nur erlaubt, eine um höchstens 1 Bit längere Eingabe in einer bestimmten Zeit zu bearbeiten. Bei einem Linearzeit-Algorithmus hingegen verdoppelt sich auch die mögliche Eingabelänge; bei einer Laufzeit von n^k vergrößert sich die mögliche Eingabelänge immerhin noch um den Faktor $\sqrt[k]{2}$. Deswegen sind Probleme, für die nur Exponentialzeit-Algorithmen existieren, praktisch nicht lösbar; daran ändert sich auch nichts Wesentliches durch die Einführung von immer schnelleren Rechnern.

Nun stellt sich natürlich sofort die Frage: Gibt es für jedes Problem einen effizienten Algorithmus? Man kann relativ leicht zeigen, dass die Antwort auf diese Frage „Nein“ ist. Die Schwierigkeit bei dieser Fragestellung liegt aber darin, dass man von vielen Problemen *nicht weiß*, ob sie effizient lösbar sind. Ganz konkret: Ein effizienter Algorithmus für das Problem 2COL ist Algorithmus 26. Ist es möglich, ebenfalls einen Polynomialzeitalgorithmus für 3COL zu finden? Viele Informatiker beschäftigen sich seit den 60er Jahren des letzten Jahrhunderts intensiv mit dieser Frage. Dabei kristallisierte sich heraus, dass viele praktisch relevante Probleme, für die kein effizienter Algorithmus bekannt ist, eine gemeinsame Eigenschaft besitzen, nämlich die der *effizienten Überprüfbarkeit* von geeigneten Lösungen. Auch 3COL gehört zu dieser Klasse von Problemen, wie sich in Kürze zeigen wird. Aber wie soll man zeigen, dass für ein Problem kein effizienter Algorithmus existiert? Nur weil kein Algorithmus bekannt ist, bedeutet das noch nicht, dass keiner existiert.

Es ist bekannt, dass die oberen Schranken (also die Laufzeit von bekannten Algorithmen) und die unteren Schranken (mindestens benötigte Laufzeit) für das Problem PARITY (und einige wenige weitere, ebenfalls sehr einfach-geartete Probleme) sehr nahe zusammen liegen. Das bedeutet also, dass nur noch unwesentliche Verbesserungen der Algorithmen für des PARITY-Problem erwartet werden können. Beim Problem 3COL ist das ganz anders: Die bekannten unteren und oberen Schranken liegen extrem weit auseinander. Deshalb ist nicht klar, ob nicht doch (extrem) bessere Algorithmen als heute bekannt im Bereich des Möglichen liegen. Aber wie untersucht man solch eine Problematik? Man müsste ja über unendlich viele Algorithmen für das Problem 3COL Untersuchungen anstellen. Dies ist äußerst schwer zu handhaben und deshalb ist der einzige bekannte Ausweg, das Problem mit einer Reihe von weiteren (aus bestimmten Gründen) interessierenden Problemen zu vergleichen und zu zeigen, dass unser zu untersuchendes Problem nicht leichter zu lösen ist als diese anderen. Hat man das geschafft, ist eine untere Schranke einer speziellen Art gefunden: Unser Problem ist nicht leichter lösbar, als

alle Probleme der Klasse von Problemen, die für den Vergleich herangezogen wurden. Nun ist aus der Beschreibung der Aufgabe aber schon klar, dass auch diese Aufgabe schwierig zu lösen ist, weil ein Problem nun mit unendlich vielen anderen Problemen zu vergleichen ist. Es zeigt sich aber, dass diese Aufgabe nicht aussichtslos ist. Bevor diese Idee weiter ausgeführt wird, soll zunächst die Klasse von Problemen untersucht werden, die für diesen Vergleich herangezogen werden sollen, nämlich die Klasse **NP**.

6.2. Effizient überprüfbare Probleme: die Klasse NP

Wie schon erwähnt, gibt es eine große Anzahl verschiedener Probleme, für die kein effizienter Algorithmus bekannt ist, die aber eine gemeinsame Eigenschaft haben: die *effiziente Überprüfbarkeit von Lösungen* für dieses Problem. Diese Eigenschaft soll an dem schon bekannten Problem 3COL veranschaulicht werden: Angenommen, man hat einen beliebigen Graphen G gegeben; wie bereits erwähnt ist kein effizienter Algorithmus bekannt, der entscheiden kann, ob der Graph G eine 3-Färbung hat (d.h., ob der fiktive Mobilfunkprovider mit 3 Funkfrequenzen auskommt). Hat man aber aus irgendwelchen Gründen eine *potenzielle* Knotenfärbung vorliegen, dann ist es leicht, diese potenzielle Knotenfärbung zu überprüfen und festzustellen, ob sie eine *korrekte* Färbung des Graphen ist, wie Algorithmus 27 zeigt.

Algorithmus 27: Ein Algorithmus zur Überprüfung einer potentiellen Färbung

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$ und eine potenzielle Knotenfärbung
Ergebnis: 1 wenn die Färbung korrekt, 0 sonst

```
begin
                                          /* Teste systematisch alle Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
                                      /* Test auf Verletzung der Knotenfärbung */
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

Das Problem 3COL hat also die Eigenschaft, dass eine potenzielle Lösung leicht daraufhin überprüft werden kann, ob sie eine tatsächliche, d.h. korrekte, Lösung ist. Viele andere praktisch relevante Probleme, für die kein effizienter Algorithmus bekannt ist, besitzen ebenfalls diese Eigenschaft. Dies soll noch an einem weiteren Beispiel verdeutlicht werden, dem so genannten *Hamiltonkreis-Problem.*

Sei wieder ein Graph $G = \big(\{v_1, \ldots v_n\}, E\big)$ gegeben. Diesmal ist eine Rundreise entlang der Kanten von G gesucht, die bei einem Knoten v_{i_1} aus G startet, wieder bei v_{i_1} endet und jeden Knoten genau einmal besucht. Genauer wird diese Rundreise im Graphen G durch eine Folge von n Knoten $(v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_{n-1}}, v_{i_n},)$ beschrieben, wobei gelten soll, dass alle Knoten $v_{i_1}, \ldots, v_{i_n}$ verschieden sind und die Kanten $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n})$ und (v_{i_n}, v_{i_1}) in G vorkommen. Eine solche Folge von Kanten wird als *Hamiltonscher Kreis* bezeichnet. Ein Hamiltonscher Kreis in einem Graphen G ist also ein Kreis, der jeden Knoten des Graphen genau einmal besucht. Das Problem, einen Hamiltonschen Kreis in einem Graphen zu finden, bezeichnet man mit HAMILTON:

PROBLEM: HAMILTON
EINGABE: Ein Graph G
FRAGE: Hat G einen Hamiltonschen Kreis?

Auch für dieses Problem ist kein effizienter Algorithmus bekannt. Aber auch hier ist offensichtlich: Bekommt man einen Graphen gegeben und eine Folge von Knoten, dann kann man sehr leicht überprüfen, ob sie ein Hamiltonscher Kreis ist – dazu ist lediglich zu testen, ob alle Knoten genau einmal besucht werden und auch alle Kanten im gegebenen Graphen vorhanden sind.

Hat man erst einmal die Beobachtung gemacht, dass viele Probleme die Eigenschaft der effizienten Überprüfbarkeit haben, ist es naheliegend, sie in einer Klasse zusammenzufassen und gemeinsam zu untersuchen. Die Hoffnung dabei ist, dass sich alle Aussagen, die man über diese Klasse herausfindet, sofort auf alle Probleme anwenden lassen. Solche Überlegungen führten zur Geburt der Klasse **NP**, in der man alle effizient überprüfbaren Probleme zusammenfasst. Aber wie kann man solch eine Klasse untersuchen? Man hat ja noch nicht einmal ein Maschinenmodell (oder eine Programmiersprache) zur Verfügung, um solch eine Eigenschaft zu modellieren. Um ein Programm für effizient überprüfbare Probleme zu schreiben, braucht man erst eine Möglichkeit, die zu überprüfenden möglichen Lösungen zu ermitteln und sie dann zu testen, d.h. man muss die Programmiersprache für **NP** in einer geeigneten Weise mit mehr „Berechnungskraft“ ausstatten.

Die erste Lösungsidee für **NP**-Probleme, nämlich alle in Frage kommenden Lösungen in einer **for**-Schleife aufzuzählen, führt zu Exponentialzeit-Lösungsalgorithmen, denn es gibt im Allgemeinen einfach so viele potenzielle Lösungen. Um erneut auf das Problem 3COL zurückzukommen: Angenommen, G ist ein Graph mit n Knoten. Dann gibt es 3^n potenzielle Färbungen, die überprüft werden müssen, denn es gibt 3 Möglichkeiten den ersten Knoten zu färben, 3 Möglichkeiten den zweiten Knoten zu färben, usw., und damit 3^n viele zu überprüfende potenzielle Färbungen. Würde man diese in einer **for**-Schleife aufzählen und auf Korrektheit testen, so führte das also zu einem Exponentialzeit-Algorithmus. Auf der anderen Seite gibt es aber Probleme, die in Exponentialzeit gelöst werden können, aber nicht zu der Intuition der effizienten Überprüfbarkeit der Klasse **NP** passen. Das Berechnungsmodell für **NP** kann also nicht einfach so gewonnen werden, dass exponentielle Laufzeit zugelassen wird, denn damit wäre man über das Ziel hinausgeschossen.

Hätte man einen Parallelrechner zur Verfügung mit so vielen Prozessoren wie es potenzielle Lösungen gibt, dann könnte man das Problem schnell lösen, denn jeder Prozessor kann unabhängig von allen anderen Prozessoren eine potenzielle Färbung überprüfen. Es zeigt sich aber, dass auch dieses Berechnungsmodell zu mächtig wäre. Es gibt Probleme, die wahrscheinlich nicht im obigen Sinne effizient überprüfbar sind, aber mit solch einem Parallelrechner trotzdem (effizient) gelöst werden könnten. In der Praxis würde uns ein derartiger paralleler Algorithmus auch nichts nützen, da man einen Rechner mit exponentiell vielen Prozessoren, also enormem Hardwareaufwand, zu konstruieren hätte. Also muss auch dieses Berechnungsmodell wieder etwas schwächer gemacht werden.

Eine Abschwächung der gerade untersuchten Idee des Parallelrechners führt zu folgendem Vorgehen: Man „rät“ für den ersten Knoten eine beliebige Farbe, dann für den zweiten Knoten auch wieder eine beliebige Farbe, solange bis für den letzten Knoten eine Farbe gewählt wurde. Danach überprüft man die geratene Färbung und akzeptiert die Eingabe, wenn die geratene Färbung eine korrekte Knotenfärbung ist. Die Eingabe ist eine positive Eingabeinstanz des **NP**-Problems 3COL, falls es eine potenzielle Lösung (Färbung) gibt, die sich bei der Überprüfung als korrekt herausstellt, d.h. im beschriebenen Rechnermodell: falls es eine Möglichkeit zu raten gibt, sodass am Ende akzeptiert (Wert 1 ausgegeben) wird. Man kann also die Berechnung durch einen Baum mit 3-fachen Verzweigungen darstellen (vgl. Abbildung 41). An den Kanten des Baumes

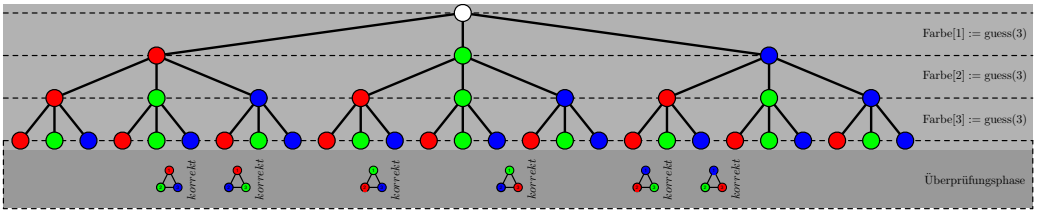


Abbildung 41: Ein Berechnungsbaum für das 3COL-Problem

findet sich das Resultat der Rateanweisung der darüberliegenden Verzweigung. Jeder Pfad in diesem sogenannten *Berechnungsbaum* entspricht daher einer Folge von Farbzuordnungen an die Knoten, d.h. einer potenziellen Färbung. Der Graph ist 3-färbbar, falls sich auf mindestens einem Pfad eine korrekte Färbung ergibt, falls also auf mindestens einem Pfad die Überprüfungsphase erfolgreich ist; der Beispielgraph besitzt sechs korrekte 3-Färbungen, ist also eine positive Instanz des 3COL-Problems.

Eine weitere, vielleicht intuitivere Vorstellung für die Arbeitsweise dieser **NP**-Maschine ist die, dass bei jedem Ratevorgang 3 verschiedene unabhängige Prozesse gestartet werden, die aber nicht miteinander kommunizieren dürfen. In diesem Sinne hat man es hier mit einem eingeschränkten Parallelrechner zu tun: Beliebige Aufspaltung (fork) ist erlaubt, aber keine Kommunikation zwischen den Prozessen ist möglich. Würde man Kommunikation zulassen, hätte man erneut den allgemeinen Parallelrechner mit exponentiell vielen Prozessoren von oben, der sich ja als zu mächtig für **NP** herausgestellt hat.

Es hat sich also gezeigt, dass eine Art „Rateanweisung“ benötigt wird. In der Programmiersprache für **NP** verwendet man dazu das neue Schlüsselwort **guess**(m), wobei m die Anzahl von Möglichkeiten ist, aus denen eine geraten wird, und legt fest, dass auch die Anweisung **guess**(m) nur einen Takt Zeit für ihre Abarbeitung benötigt. Berechnungen, die, wie soeben beschrieben, verschiedene Möglichkeiten raten können, heißen *nichtdeterministisch*. Es sei wiederholt, dass *festgelegt* (definiert) wird, dass ein nichtdeterministischer Algorithmus bei einer Eingabe den Wert 1 berechnet, falls *eine Möglichkeit* geraten werden kann, sodass der Algorithmus auf die Anweisung „**return** 1“ stößt. Die Klasse **NP** umfasst nun genau die Probleme, die von nichtdeterministischen Algorithmen mit polynomieller Laufzeit gelöst werden können. „**NP**“ steht dabei für „**n**ichtdeterministische **P**olynomialzeit“, nicht etwa, wie mitunter zu lesen, für „Nicht-Polynomial-zeit“. (Eine formale Präsentation der Äquivalenz zwischen effizienter Überprüfbarkeit und Polynomialzeit in der **NP**-Programmiersprache findet sich z.B. in [MD79, Kapitel 2.3].)

Mit Hilfe eines nichtdeterministischen Algorithmus kann das 3COL-Problem in Polynomialzeit gelöst werden (siehe Algorithmus 28). Die zweite Phase von Algorithmus 28, die *Überprüfungsphase*, entspricht dabei genau dem oben angegebenen Algorithmus zum effizienten Überprüfen von möglichen Lösungen des 3COL-Problems (vgl. Algorithmus 27).

Dieser nichtdeterministische Algorithmus läuft in Polynomialzeit, denn man benötigt für einen Graphen mit n Knoten mindestens n Bits, um ihn zu speichern (kodieren), und der Algorithmus braucht im schlechtesten Fall $O(n)$ (Ratephase) und $O(n^2)$ (Überprüfungsphase), also insgesamt $O(n^2)$ Takte Zeit. Damit ist gezeigt, dass 3COL in der Klasse **NP** enthalten ist, denn es wurde ein nichtdeterministischer Polynomialzeitalgorithmus gefunden, der 3COL löst. Ebenso einfach könnte man nun einen nichtdeterministischen Polynomialzeitalgorithmus entwickeln, der das Problem HAMILTON löst: Der Algorithmus wird in einer ersten Phase eine Knotenfolge raten und dann in einer zweiten Phase überprüfen, dass die Bedingungen, die an einen Hamiltonschen Kreis gestellt werden, bei der geratenen Folge erfüllt sind. Dies zeigt, dass auch HAMILTON in der Klasse **NP** liegt.

Algorithmus 28: Ein nichtdeterministischer Algorithmus für 3COL

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$
Ergebnis: 1 wenn eine Färbung existiert, 0 sonst

```
begin
    /* Ratephase */
    for (i = 1 to n) do
        Farbe[i] = guess(3);
    end
    /* Überprüfungsphase */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

Dass eine nichtdeterministische Maschine nicht gebaut werden kann, spielt hier keine Rolle. Nichtdeterministische Berechnungen sollen hier lediglich als Gedankenmodell für unsere Untersuchungen herangezogen werden, um Aussagen über die (Nicht-) Existenz von effizienten Algorithmen machen zu können.

6.3. Schwierigste Probleme in NP: der Begriff der NP-Vollständigkeit

Es ist nun klar, was es bedeutet, dass ein Problem in **NP** liegt. Es liegt aber auch auf der Hand, dass alle Probleme aus **P** auch in **NP** liegen, da bei der Einführung von **NP** ja nicht verlangt wurde, dass die **guess**-Anweisung verwendet werden muss. Damit ist jeder deterministische Algorithmus automatisch auch ein (eingeschränkter) nichtdeterministischer Algorithmus. Nun ist aber auch schon bekannt, dass es Probleme in **NP** gibt, z.B. 3COL und weitere Probleme, von denen nicht bekannt ist, ob sie in **P** liegen. Das führt zu der Vermutung, dass $\mathbf{P} \neq \mathbf{NP}$.

Es gibt also in **NP** anscheinend unterschiedlich schwierige Probleme: einerseits die **P**-Probleme (also die leichten Probleme), und andererseits die Probleme, von denen man nicht weiß, ob sie in **P** liegen (die schweren Probleme). Es liegt also nahe, eine allgemeine Möglichkeit zu suchen, Probleme in **NP** bezüglich ihrer Schwierigkeit zu vergleichen. Ziel ist, wie oben erläutert, eine Art von unterer Schranke für Probleme wie 3COL: Es soll gezeigt werden, dass 3COL mindestens so schwierig ist, wie jedes andere Problem in **NP**, also in gewissem Sinne ein *schwierigstes Problem in* **NP** ist.

Für diesen Vergleich der Schwierigkeit ist die erste Idee natürlich, einfach die Laufzeit von (bekannten) Algorithmen für das Problem heranzuziehen. Dies ist jedoch nicht erfolgversprechend, denn was soll eine „größte“ Laufzeit sein, die Programme für „schwierigste“ Probleme in **NP** ja haben müssten? Außerdem hängt die Laufzeit eines Algorithmus vom verwendeten Berechnungsmodell ab. So kennen Turingmaschinen keine Arrays im Gegensatz zu der hier verwendeten C-Variante. Also würde jeder Algorithmus, der Arrays verwendet, auf einer Turingmaschine mühsam simuliert werden müssen und damit langsamer abgearbeitet werden, als bei einer Hochsprache, die Arrays enthält. Obwohl sich die Komplexität eines Problems nicht ändert, würde man sie verschieden messen, je nachdem welches Berechnungsmodell verwendet würde.

Ein weiterer Nachteil dieses Definitionsversuchs wäre es, dass die Komplexität (Schwierigkeit) eines Problems mit bekannten Algorithmen gemessen würde. Das würde aber bedeuten, dass jeder neue und schnellere Algorithmus Einfluss auf die Komplexität hätte, was offensichtlich so keinen Sinn macht. Aus diesen und anderen Gründen führt die erste Idee nicht zum Ziel.

Eine zweite, erfolgversprechendere Idee ist die folgende: Ein Problem A ist nicht (wesentlich) schwieriger als ein Problem B, wenn man A mit der Hilfe von B (als Unterprogramm) effizient lösen kann. Ein einfaches Beispiel ist die Multiplikation von n Zahlen. Angenommen, man hat schon ein Programm, dass zwei Zahlen multiplizieren kann; dann ist es nicht wesentlich schwieriger, auch n Zahlen zu multiplizieren, wenn die Routine für die Multiplikation von zwei Zahlen verwendet wird. Dieser Ansatz ist unter dem Namen *relative Berechenbarkeit* bekannt, der genau den oben beschriebenen Sachverhalt widerspiegelt: Multiplikation von n Zahlen (so genannte *iterierte Multiplikation*) ist relativ zur Multiplikation zweier Zahlen (leicht) berechenbar.

Da das Prinzip der relativen Berechenbarkeit so allgemein gehalten ist, gibt es innerhalb der theoretischen Informatik sehr viele verschiedene Ausprägungen dieses Konzepts. Für die **P-NP**-Problematik ist folgende Version der relativen Berechenbarkeit, d.h. die folgende Art von erlaubten „Unterprogrammaufrufen“, geeignet:
Seien zwei Probleme A und B gegeben. Das Problem A ist nicht schwerer als B, falls es eine effizient zu berechnende Transformation T gibt, die Folgendes leistet: Wenn x eine Eingabeinstanz von Problem A ist, dann ist $T(x)$ eine Eingabeinstanz für B. Weiterhin gilt: x ist *genau dann* eine positive Instanz von A (d.h. ein Entscheidungsalgorithmus für A muss den Wert 1 für Eingabe x liefern), wenn $T(x)$ eine positive Instanz von Problem B ist. Erneut soll „effizient berechenbar“ hier bedeuten: in Polynomialzeit berechenbar. Es muss also einen Polynomialzeitalgorithmus geben, der die Transformation T ausführt. Das Entscheidungsproblem A ist damit effizient transformierbar in das Problem B. Man sagt auch: A ist reduzierbar auf B; oder intuitiver: A ist nicht schwieriger als B, oder B ist mindestens so schwierig wie A. Formal schreibt man dann $A \leq B$.

Um für dieses Konzept ein wenig mehr Intuition zu gewinnen, sei erwähnt, dass man sich eine solche Transformation auch wie folgt vorstellen kann: A lässt sich auf B reduzieren, wenn ein Algorithmus für A angegeben werden kann, der ein Unterprogramm U_B für B genau so verwendet wie in Algorithmus 29 gezeigt.

Dabei ist zu beachten, dass das Unterprogramm für B nur genau einmal und zwar am Ende aufgerufen werden darf. Das Ergebnis des Algorithmus für A ist genau das Ergebnis, das dieser Unterprogrammaufruf liefert. Es gibt zwar, wie oben erwähnt, auch allgemeinere Ausprägungen der relativen Berechenbarkeit, die diese Einschränkung nicht haben, diese sind aber für die folgenden Untersuchungen nicht relevant.

Nachdem nun ein Vergleichsbegriff für die Schwierigkeit von Problemen aus **NP** gefunden wurde, kann auch definiert werden, was unter einem „schwierigsten“ Problem in **NP** zu verstehen ist. Ein Problem C ist ein schwierigstes Problem in **NP**, wenn alle anderen Probleme in **NP** höchstens so schwer wie C sind. Formaler ausgedrückt sind dazu zwei Eigenschaften von C nachzuweisen:

(1) C ist ein Problem aus **NP**.

(2) C ist mindestens so schwierig wie jedes andere **NP**-Problem A; d.h.: für alle Probleme A aus **NP** gilt: $A \leq C$.

Solche schwierigsten Probleme in **NP** sind unter der Bezeichnung **NP**-*vollständige Probleme* bekannt. Nun sieht die Aufgabe, von einem Problem zu zeigen, dass es **NP**-vollständig ist, ziemlich hoffnungslos aus. Immerhin ist zu zeigen, dass für alle Probleme aus **NP** – und damit unendlich viele – gilt, dass sie höchstens so schwer sind wie das zu untersuchende Problem,

Algorithmus 29: Algorithmische Darstellung der Benutzung einer Reduktionsfunktion

Eingabe: Instanz x für das Problem A
Ergebnis: 1 wenn $x \in A$ und 0 sonst

begin
  /* T ist die Reduktionsfunktion (polynomialzeitberechenbar) */
  berechne $y = T(x)$;
  /* y ist Instanz des Problems B */
  $z = U_B(y)$;
  /* z ist 1 genau dann, wenn $x \in A$ gilt */
  return z;
end

und damit scheint man der Schwierigkeit beim Nachweis unterer Schranken nicht entgangen zu sein. Dennoch konnten der russische Mathematiker Leonid Levin und der amerikanische Mathematiker Stephen Cook Anfang der siebziger Jahre des letzten Jahrhunderts unabhängig voneinander die Existenz von solchen **NP**-vollständigen Problemen zeigen. Hat man nun erst einmal *ein* solches Problem identifiziert, ist die Aufgabe, *weitere* **NP**-vollständige Probleme zu finden, wesentlich leichter. Dies ist sehr leicht einzusehen: Ein **NP**-Problem C ist ein schwierigstes Problem in **NP**, wenn es ein anderes schwierigstes Problem B gibt, sodass C nicht leichter als B ist. Das führt zu folgendem „Kochrezept“:

Nachweis der NP-Vollständigkeit eines Problems C:

i) Zeige, dass C in **NP** enthalten ist, indem dafür ein geeigneter nichtdeterministischer Polynomialzeitalgorithmus konstruiert wird.

ii) Suche ein geeignetes „ähnliches“ schwierigstes Problem B in **NP** und zeige, dass C nicht leichter als B ist. Formal: Finde ein **NP**-vollständiges Problem B und zeige $B \leq C$ mit Hilfe einer geeigneten Transformation T.

Den zweiten Schritt kann man oft relativ leicht mit Hilfe von bekannten Sammlungen **NP**-vollständiger Problemen erledigen. Das Buch von Garey und Johnson [MD79] ist eine solche Sammlung (siehe auch die Abbildungen 43 und 44), die mehr als 300 **NP**-vollständige Probleme enthält. Dazu wählt man ein möglichst ähnliches Problem aus und versucht dann eine geeignete Reduktionsfunktion für das zu untersuchende Problem zu finden.

6.3.1. Traveling Salesperson ist NP-vollständig

Wie kann man zeigen, dass Traveling Salesperson **NP**-vollständig ist? Dazu wird zuerst die genaue Definition dieses Problems benötigt:

Problem:	TRAVELING SALESPERSON (TSP)
Eingabe:	Eine Menge von Städten $C = \{c_1, \ldots, c_n\}$ und eine $n \times n$ Entfernungsmatrix D, wobei das Element $D[i, j]$ der Matrix D die Entfernung zwischen Stadt c_i und c_j angibt. Weiterhin eine Obergrenze $k \geq 0$ für die maximal erlaubte Länge der Tour
Frage:	Gibt es eine Rundreise, die einerseits alle Städte besucht, aber andererseits eine Gesamtlänge von höchstens k hat?

Nun zum ersten Schritt des Nachweises der **NP**-Vollständigkeit von TSP: Offensichtlich gehört auch das Traveling Salesperson Problem zur Klasse **NP**, denn man kann nichtdeterministisch

Aus dem Graphen G links berechnet die Transformation die rechte Eingabe für das TSP. Die dick gezeichneten Verbindungen deuten eine Entfernung von 1 an, wogegen dünne Linien eine Entfernung von 6 symbolisieren. Weil G den Hamiltonkreis $1, 2, 3, 4, 5, 1$ hat, gibt es rechts eine Rundreise $1, 2, 3, 4, 5, 1$ mit Gesamtlänge 5.

Im Gegensatz dazu berechnet die Transformation hier aus dem Graphen G' auf der linken eine Eingabe für das TSP auf der rechten Seite, die, wie man sich leicht überzeugt, keine Rundreise mit einer maximalen Gesamtlänge von 5 hat. Dies liegt daran, dass der ursprüngliche Graph G' keinen Hamiltonschen Kreis hatte.

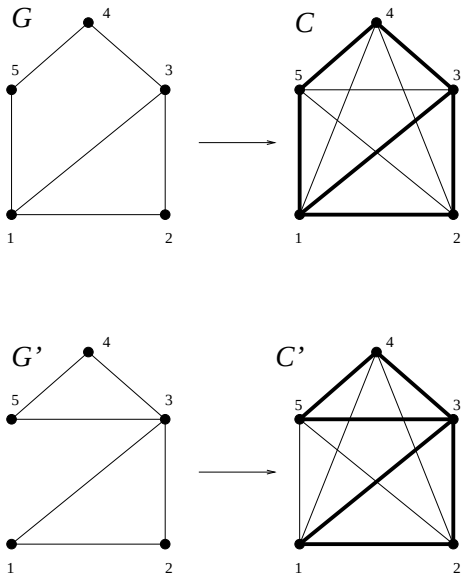


Abbildung 42: Beispiele für die Wirkungsweise von Algorithmus 30

eine Folge von n Städten raten (eine potenzielle Rundreise) und dann leicht überprüfen, ob diese potenzielle Tour durch alle Städte verläuft und ob die zurückzulegende Entfernung maximal k beträgt. Ein entsprechender nichtdeterministischer Polynomialzeitalgorithmus ist leicht zu erstellen. Damit ist der erste Schritt zum Nachweis der **NP**-Vollständigkeit von TSP getan und Punkt (1) des „Kochrezepts“ abgehandelt.

Als nächstes (Punkt (2)) soll von einem anderen **NP**-vollständigen Problem gezeigt werden, dass es effizient in TSP transformiert werden kann. Geeignet dazu ist das im Text betrachtete Hamitonkreis-Problem, das bekanntermaßen **NP**-vollständig ist. Es ist also zu zeigen: HAMILTON $\leq$ TSP.

Folgende Idee führt zum Ziel: Gegeben ist eine Instanz $G = (V, E)$ von HAMILTON. Transformiere G in folgende Instanz von TSP: Als Städtemenge C wählen wir die Knoten V des Graphen G. Die Entfernungen zwischen den Städten sind definiert wie folgt: $D[i, j] = 1$, falls es in E eine Kante von Knoten i zu Knoten j gibt, ansonsten setzt man $D[i, j]$ auf einen sehr großen Wert, also z.B. $n + 1$, wenn n die Anzahl der Knoten von G ist. Dann gilt klarerweise: Wenn G einen Hamiltonschen Kreis besitzt, dann ist der gleiche Kreis eine Rundreise in C mit Gesamtlänge n. Wenn G keinen Hamiltonschen Kreis besitzt, dann kann es keine Rundreise durch die Städte C mit Länge höchstens n geben, denn jede Rundreise muss mindestens eine Strecke von einer Stadt i nach einer Stadt j zurücklegen, die keiner Kante in G entspricht (denn ansonsten hätte G ja einen Hamiltonschen Kreis). Diese einzelne Strecke von i nach j hat dann aber schon Länge $n + 1$ und damit ist eine Gesamtlänge von n oder weniger nicht mehr erreichbar. Die Abbildung 42 zeigt zwei Beispiele für die Wirkungsweise der Transformation, die durch Algorithmus 30 in Polynomialzeit berechnet wird.

6.4. Die Auswirkungen der NP-Vollständigkeit

Welche Bedeutung haben nun die **NP**-vollständigen Probleme für die Klasse **NP**? Könnte jemand einen deterministischen Polynomialzeitalgorithmus $\mathcal{A}_C$ für ein **NP**-vollständiges Problem C angeben, dann hätte man für jedes **NP**-Problem einen Polynomialzeitalgorithmus gefunden

Algorithmus 30: Ein Algorithmus für die Reduktion von HAMILTON auf TSP

Eingabe: Graph $G = (V, E)$, wobei $V = \{1, \ldots, n\}$
Ergebnis: Eine Instanz (C, D, k) für TSP

```
begin
    /* Die Knoten entsprechen den Städten */
    C = V;
    /* Überprüfe alle potentiell existierenden Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if ((v_i, v_j) ∈ E) then
                /* Kanten entsprechen kleinen Entfernungen */
                D[i][j] = 1;
            else
                /* nicht existierende Kante, dann sehr große Entfernung */
                D[i][j] = n + 1;
            end
        end
    end
    /* Gesamtlänge k der Rundreise ist Anzahl der Städte n */
    k = n;
    /* Gebe die berechnete TSP-Instanz zurück */
    return (C, D, k);
end
```

(d.h. **P** = **NP**). Diese überraschende Tatsache lässt sich leicht einsehen, denn für jedes Problem A aus **NP** gibt es eine Transformation T mit der Eigenschaft, dass x genau dann eine positive Eingabeinstanz von A ist, wenn $T(x)$ eine positive Instanz von C ist. Damit löst Algorithmus 31 das Problem A in Polynomialzeit. Es gilt also: Ist irgendein **NP**-vollständiges Problem effizient lösbar, dann ist **P** = **NP**.

Algorithmus 31: Ein fiktiver Algorithmus für Problem A

Eingabe: Instanz x für das Problem A
Ergebnis: `true`, wenn $x \in A$, `false` sonst

```
begin
    /* T ist die postulierte Reduktionsfunktion */
    y = T(x);
    z = A_C(y);
    return z;
end
```

Sei nun angenommen, dass jemand **P** ≠ **NP** gezeigt hat. In diesem Fall ist aber auch klar, dass dann für kein **NP**-vollständiges Problem ein (effizienter) Polynomialzeitalgorithmus existieren kann, denn sonst würde sich ja der Widerspruch **P** = **NP** ergeben. Ist das Problem C also **NP**-vollständig, so gilt: C hat genau dann einen effizienten Algorithmus, wenn **P** = **NP**, also wenn jedes Problem in **NP** einen effizienten Algorithmus besitzt. Diese Eigenschaft macht die **NP**-vollständigen Probleme für die Theoretiker so interessant, denn eine Klasse von unendlich

vielen Problemen kann untersucht werden, indem man nur ein einziges Problem betrachtet. Man kann sich das auch wie folgt vorstellen: Alle relevanten Eigenschaften aller Probleme aus **NP** wurden in ein einziges Problem „destilliert“. Die **NP**-vollständigen Probleme sind also in diesem Sinn *prototypische* **NP**-Probleme.

Trotz intensiver Bemühungen in den letzten 30 Jahren konnte bisher niemand einen Polynomialzeitalgorithmus für ein **NP**-vollständiges Problem finden. Dies ist ein Grund dafür, dass man heute $\mathbf{P} \neq \mathbf{NP}$ annimmt. Leider konnte auch dies bisher nicht gezeigt werden, aber in der theoretischen Informatik gibt es starke Indizien für die Richtigkeit dieser Annahme, sodass heute die große Mehrheit der Forscher von $\mathbf{P} \neq \mathbf{NP}$ ausgeht.

Für die Praxis bedeutet dies Folgendes: Hat man von einem in der Realität auftretenden Problem gezeigt, dass es **NP**-vollständig ist, dann kann man getrost aufhören, einen effizienten Algorithmus zu suchen. Wie wir ja gesehen haben, kann ein solcher nämlich (zumindest unter der gut begründbaren Annahme $\mathbf{P} \neq \mathbf{NP}$) nicht existieren.

Nun ist auch eine Antwort für das 3COL-Problem gefunden. Es wurde gezeigt [MD79], dass kCOL für $k \geq 3$ **NP**-vollständig ist. Der fiktive Mobilfunkplaner hat also Pech gehabt: Es ist unwahrscheinlich, dass er jemals ein korrektes effizientes Planungsverfahren finden wird.

Ein **NP**-Vollständigkeitsnachweis eines Problems ist also ein starkes Indiz für seine praktische Nicht-Handhabbarkeit. Auch die **NP**-Vollständigkeit eines Problems, das mit dem Spiel *Minesweeper* zu tun hat, bedeutet demnach lediglich, dass dieses Problem höchstwahrscheinlich nicht effizient lösbar sein wird. Ein solcher Vollständigkeitsbeweis hat nichts mit einem Schritt in Richtung auf eine Lösung des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems zu tun, wie irreführenderweise gelegentlich zu lesen ist. Übrigens ist auch für eine Reihe weiterer Spiele ihre **NP**-Vollständigkeit bekannt. Dazu gehören u.a. bestimmte Puzzle- und Kreuzwortspiele. Typische Brettspiele, wie Dame, Schach oder GO, sind hingegen (verallgemeinert auf Spielbretter der Größe $n \times n$) **PSPACE**-vollständig. Die Klasse **PSPACE** ist eine noch deutlich mächtigere Klasse als **NP**. Damit sind also diese Spiele noch viel komplexer als Minesweeper und andere **NP**-vollständige Probleme.

6.5. Der Umgang mit NP-vollständigen Problemen in der Praxis

Viele in der Praxis bedeutsame Probleme sind **NP**-vollständig (vgl. die Abbildungen 43 und 44). Ein Anwendungsentwickler wird es aber sicher schwer haben, seinem Management mitteilen zu müssen, dass ein aktuelles Projekt nicht durchgeführt werden kann, weil keine geeigneten Algorithmen zur Verfügung stehen (Wahrscheinlich würden in diesem Fall einfach „geeignetere“ Entwickler eingestellt werden!). Es stellt sich daher also die Frage, wie man mit solchen **NP**-vollständigen Problemen in der Praxis umgeht. Zu dieser Fragestellung hat die theoretische Informatik ein ausgefeiltes Instrumentarium entwickelt.

Eine erste Idee wäre es, sich mit Algorithmen zufrieden zu geben, die mit Zufallszahlen arbeiten und die nur mit sehr großer Wahrscheinlichkeit die richtige Lösung berechnen, aber sich auch mit kleiner (vernachlässigbarer) Wahrscheinlichkeit irren dürfen. Solche Algorithmen sind als *probabilistische* oder *randomisierte Algorithmen* bekannt [RP95] und werden beispielsweise in der Kryptographie mit sehr großem Erfolg angewendet. Das prominenteste Beispiel hierfür sind Algorithmen, die testen, ob eine gegebene Zahl eine Primzahl ist und sich dabei fast nie irren. Primzahlen spielen bekanntermaßen im RSA-Verfahren und damit bei PGP und ähnlichen Verschlüsselungen eine zentrale Rolle. Es konnte aber gezeigt werden, dass probabilistische Algorithmen uns bei den **NP**-vollständigen Problemen wohl nicht weiterhelfen. So weiß man heute, dass die Klasse der Probleme, die sich mit probabilistischen Algorithmen effizient lösen lässt, höchstwahrscheinlich nicht die Klasse **NP** umfasst. Deshalb liegen (höchstwahrscheinlich) insbesondere alle **NP**-vollständigen Probleme außerhalb der Möglichkeiten von effizienten probabilistischen Algorithmen.

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [MD79].

PROBLEM: CLUSTER [GT19]
EINGABE: Netzwerk $G = (V, E)$, positive Integerzahl K
FRAGE: Gibt es eine Menge von mindestens K Knoten, die paarweise miteinander verbunden sind?

PROBLEM: NETZ-AUFTEILUNG [ND16]
EINGABE: Netzwerk $G = (V, E)$, Kapazität für jede Kante in E, positive Integerzahl K
FRAGE: Kann man das Netzwerk so in zwei Teile zerlegen, dass die Gesamtkapazität aller Verbindungen zwischen den beiden Teilen mindestens K beträgt?

PROBLEM: NETZ-REDUNDANZ [ND18]
EINGABE: Netzwerk $G = (V, E)$, Kosten für Verbindungen zwischen je zwei Knoten aus V, Budget B
FRAGE: Kann G so um Verbindungen erweitert werden, dass zwischen je zwei Knoten mindestens zwei Pfade existieren und die Gesamtkosten für die Erweiterung höchstens B betragen?

PROBLEM: OBJEKTE SPEICHERN [SR1]
EINGABE: Eine Menge U von Objekten mit Speicherbedarf $s(u)$ für jedes $u \in U$; Kachelgröße S, positive Integerzahl K
FRAGE: Können die Objekte in U auf K Kacheln verteilt werden?

PROBLEM: DATENKOMPRESSION [SR8]
EINGABE: Endliche Menge R von Strings über festgelegtem Alphabet, positive Integerzahl K
FRAGE: Gibt es einen String S der Länge höchstens K, sodass jeder String aus R als Teilfolge von S vorkommt?

PROBLEM: K-SCHLÜSSEL [SR26]
EINGABE: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, positive Integerzahl K
FRAGE: Gibt es einen Schlüssel mit höchstens K Attributen?

PROBLEM: BCNF [SR29]
EINGABE: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, Teilmenge $A' \subseteq A$
FRAGE: Verletzt die Menge A' die Boyce-Codd-Normalform?

PROBLEM: MP-SCHEDULE [SS8]
EINGABE: Menge T von Tasks, Länge für jede Task, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T mit Ausführungszeit höchstens D?

PROBLEM: PREEMPT-SCHEDULE [SS12]
EINGABE: Menge T von Tasks, Länge für jede Task, Präzedenzrelation auf den Tasks, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T, das die Präzedenzrelationen berücksichtigt und Ausführungszeit höchstens D hat?

PROBLEM: DEADLOCK [SS22]
EINGABE: Menge von Prozessen, Menge von Ressourcen, aktuelle Zustände der Prozesse und aktuell allokierte Ressourcen
FRAGE: Gibt es einen Kontrollfluss, der zum Deadlock führt?

PROBLEM: K-REGISTER [PO3]
EINGABE: Menge V von Variablen, die in einer Schleife benutzt werden, für jede Variable einen Gültigkeitsbereich, positive Integerzahl K
FRAGE: Können die Schleifenvariablen mit höchstens K Registern gespeichert werden?

PROBLEM: REKURSION [PO20]
EINGABE: Menge A von Prozedur-Identifiern, Pascal-Programmfragment mit Deklarationen und Aufrufen der Prozeduren aus A
FRAGE: Ist eine der Prozeduren aus A formal rekursiv?

Abbildung 43: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 1)

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [MD79].

Problem:	LR(K)-GRAMMATIK [AL15]
Eingabe:	Kontextfreie Grammatik G, positive Integerzahl K (unär)
Frage:	Ist die Grammatik G nicht LR(K)?
Problem:	ZWANGSBEDINGUNG [LO5]
Eingabe:	Menge von Booleschen Constraints, positive Integerzahl K
Frage:	Können mindestens K der Constraints gleichzeitig erfüllt werden?
Problem:	INTEGER PROGRAM [MP1]
Eingabe:	Lineares Programm
Frage:	Hat das Programm eine Lösung, die nur ganzzahlige Werte enthält?
Problem:	KREUZWORTRÄTSEL [GP15]
Eingabe:	Menge W von Wörtern, Gitter mit schwarzen und weißen Feldern
Frage:	Können die weißen Felder des Gitters mit Wörtern aus W gefüllt werden?

Abbildung 44: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 2)

Nun könnte man auch versuchen, „exotischere“ Computer zu bauen. In der letzten Zeit sind zwei potenzielle Auswege bekannt geworden: DNA-Computer und Quantencomputer.

Es konnte gezeigt werden, dass DNA-Computer (siehe [Pău98]) jedes **NP**-vollständige Problem in Polynomialzeit lösen können. Für diese Berechnungsstärke hat man aber einen Preis zu zahlen: Die Anzahl und damit die Masse der DNA-Moleküle, die für die Berechnung benötigt werden, wächst exponentiell in der Eingabelänge. Das bedeutet, dass schon bei recht kleinen Eingaben mehr Masse für eine Berechnung gebraucht würde, als im ganzen Universum vorhanden ist. Bisher ist kein Verfahren bekannt, wie dieses Masseproblem gelöst werden kann, und es sieht auch nicht so aus, als ob es gelöst werden kann, wenn $\mathbf{P} \neq \mathbf{NP}$ gilt. Dieses Problem erinnert an das oben im Kontext von Parallelrechnern schon erwähnte Phänomen: Mit exponentiell vielen Prozessoren lassen sich **NP**-vollständige Probleme lösen, aber solche Parallelrechner haben natürlich explodierende Hardware-Kosten.

Der anderer Ausweg könnten Quantencomputer sein (siehe [Hom08, Gru99]). Hier scheint die Situation zunächst günstiger zu sein: Die Fortschritte bei der Quantencomputer-Forschung verlaufen immens schnell, und es besteht die berechtigte Hoffnung, dass Quantencomputer mittelfristig verfügbar sein werden. Aber auch hier sagen theoretische Ergebnisse voraus, dass Quantencomputer (höchstwahrscheinlich) keine **NP**-vollständigen Probleme lösen können. Trotzdem sind Quantencomputer interessant, denn es ist bekannt, dass wichtige Probleme existieren, für die kein Polynomialzeitalgorithmus bekannt ist und die wahrscheinlich nicht **NP**-vollständig sind, die aber auf Quantencomputern effizient gelöst werden können. Das prominenteste Beispiel hierfür ist die Aufgabe, eine ganze Zahl in ihre Primfaktoren zu zerlegen.

Die bisher angesprochenen Ideen lassen also die Frage, wie man mit **NP**-vollständigen Problemen umgeht, unbeantwortet. In der Praxis gibt es im Moment zwei Hauptansatzpunkte: Die erste Möglichkeit ist die, die Allgemeinheit des untersuchten Problems zu beschränken und eine spezielle Version zu betrachten, die immer noch für die geplante Anwendung ausreicht. Zum Beispiel sind Graphenprobleme oft einfacher, wenn man zusätzlich fordert, dass die Knoten des Graphen in der (Euklidischen) Ebene lokalisiert sind. Deshalb sollte die erste Idee bei der Behandlung von **NP**-vollständigen Problemen immer sein, zu untersuchen, welche Einschränkungen man an das Problem machen kann, ohne die praktische Aufgabenstellung zu verfälschen. Gerade diese Einschränkungen können dann effiziente Algorithmen ermöglichen.

Die zweite Möglichkeit sind sogenannte *Approximationsalgorithmen* (vgl. [GPV$^+$99]). Die Idee hier ist es, nicht die optimalen Lösungen zu suchen, sondern sich mit einem kleinen garantierten Fehler zufrieden zu geben. Dazu folgendes Beispiel. Es ist bekannt, dass das TSP auch dann noch **NP**-vollständig ist, wenn man annimmt, dass die Städte in der Euklidischen Ebene lokalisiert

sind, d.h. man kann die Städte in einer fiktiven Landkarte einzeichnen, sodass die Entfernungen zwischen den Städten proportional zu den Abständen auf der Landkarte sind. Das ist sicherlich in der Praxis keine einschränkende Abschwächung des Problems und zeigt, dass die oben erwähnte Methode nicht immer zum Erfolg führen muss: Hier bleibt auch das eingeschränkte Problem **NP**-vollständig. Aber für diese eingeschränkte TSP-Variante ist ein Polynomialzeitalgorithmus bekannt, der immer eine Rundreise berechnet, die höchstens um einen beliebig wählbaren Faktor schlechter ist, als die optimale Lösung. Ein Chip-Hersteller, der bei der Bestückung seiner Platinen die Wege der Roboterköpfe minimieren möchte, kann also beschließen, sich mit einer Tour zufrieden zu geben, die um 5 % schlechter ist als die optimale. Für dieses Problem existiert ein effizienter Algorithmus! Dieser ist für die Praxis völlig ausreichend.

⋆⋆⋆ Ende ⋆⋆⋆

A. Grundlagen und Schreibweisen

A.1. Mengen

Es ist sehr schwer den fundamentalen Begriff der Menge mathematisch exakt zu definieren. Aus diesem Grund soll uns hier die von Cantor im Jahr 1895 gegebene Erklärung genügen, da sie für unsere Zwecke völlig ausreichend ist:

Definition 72 (Georg Cantor ([Can95])): *Unter einer ‚*Menge*' verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die ‚*Elemente*' von M genannt werden) zu einem Ganzen*[19].

Für die Formulierung „genau dann wenn" verwenden wir im Folgenden die Abkürzung gdw. um Schreibarbeit zu sparen.

A.1.1. Die Elementbeziehung und die Enthaltenseinsrelation

Sehr oft werden einfache große lateinische Buchstaben wie N, M, A, B oder C als Symbole für Mengen verwendet und kleine Buchstaben für die Elemente einer Menge. Mengen von Mengen notiert man gerne mit kalligraphischen Buchstaben wie $\mathcal{A}$, $\mathcal{B}$ oder $\mathcal{M}$.

Definition 73: *Sei M eine beliebige Menge, dann ist*

- $a \in M$ *gdw. a ist ein Element der Menge M,*
- $a \notin M$ *gdw. a ist kein Element der Menge M,*
- $M \subseteq N$ *gdw. aus $a \in M$ folgt $a \in N$ (M ist* Teilmenge *von N),*
- $M \not\subseteq N$ *gdw. es gilt nicht $M \subseteq N$. Gleichwertig: es gibt ein $a \in M$ mit $a \notin N$ (M ist keine Teilmenge von N) und*
- $M \subset N$ *gdw. es gilt $M \subseteq N$ und $M \neq N$ (M ist echte Teilmenge von N).*

Statt $a \in M$ schreibt man auch $M \ni a$, was in einigen Fällen zu einer deutlichen Vereinfachung der Notation führt.

A.1.2. Definition spezieller Mengen

Spezielle Mengen können auf verschiedene Art und Weise definiert werden, wie z.B.

- durch Angabe von Elementen: So ist $\{a_1, \ldots, a_n\}$ die Menge, die aus den Elementen a_1, $\ldots$, a_n besteht, oder
- durch eine Eigenschaft E: Dabei ist $\{a \mid E(a)\}$ die Menge aller Elemente a, die die Eigenschaft[20] E besitzen.

Alternativ zu der Schreibweise $\{a \mid E(a)\}$ wird auch oft $\{a\colon E(a)\}$ verwendet.

[19] Diese Zitat entspricht der originalen Schreibweise von Cantor.
[20] Die Eigenschaft E kann man dann auch als *Prädikat* bezeichnen.

Beispiel 74: *Mengen, die durch die Angabe von Elementen definiert sind:*

- $\mathbb{B} =_{\text{def}} \{0, 1\}$
- $\mathbb{N} =_{\text{def}} \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$ *(Menge der* natürlichen Zahlen*)*
- $\mathbb{Z} =_{\text{def}} \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots\}$ *(Menge der* ganzen Zahlen*)*
- $2\mathbb{Z} =_{\text{def}} \{0, \pm 2, \pm 4, \pm 6, \pm 8, \dots\}$ *(Menge der geraden ganzen Zahlen)*
- $\mathbb{P} =_{\text{def}} \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$ *(Menge der* Primzahlen*)*

Beispiel 75: *Mengen, die durch eine Eigenschaft E definiert sind:*

- $\{n \mid n \in \mathbb{N}$ *und* n *ist durch* 3 *teilbar*$\}$
- $\{n \mid n \in \mathbb{N}$ *und* n *ist Primzahl und* $n \leq 40\}$
- $\emptyset =_{\text{def}} \{a \mid a \neq a\}$ *(die leere Menge)*

Aus Definition 73 ergibt sich, dass die leere Menge (Schreibweise: $\emptyset$) Teilmenge jeder Menge ist. Dabei ist zu beachten, dass $\{\emptyset\} \neq \emptyset$ gilt, denn $\{\emptyset\}$ enthält *ein* Element (die leere Menge) und $\emptyset$ enthält *kein* Element.

A.1.3. Operationen auf Mengen

Definition 76: *Seien A und B beliebige Mengen, dann ist*

- $A \cap B =_{\text{def}} \{a \mid a \in A$ *und* $a \in B\}$ *(*Schnitt *von A und B),*
- $A \cup B =_{\text{def}} \{a \mid a \in A$ *oder* $a \in B\}$ *(*Vereinigung *von A und B),*
- $A \setminus B =_{\text{def}} \{a \mid a \in A$ *und* $a \notin B\}$ *(*Differenz *von A und B),*
- $\overline{A} =_{\text{def}} M \setminus A$ *(*Komplement *von A bezüglich einer festen Grundmenge M) und*
- $\mathcal{P}(A) =_{\text{def}} \{B \mid B \subseteq A\}$ *(*Potenzmenge *von A).*

Zwei Mengen A und B mit $A \cap B = \emptyset$ *nennt man* disjunkt*.*

Beispiel 77: *Sei* $A = \{2, 3, 5, 7\}$ *und* $B = \{1, 2, 4, 6\}$*, dann ist* $A \cap B = \{2\}$*,* $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ *und* $A \setminus B = \{3, 5, 7\}$*. Wählen wir als Grundmenge die natürlichen Zahlen, also* $M = \mathbb{N}$*, dann ist* $\overline{A} = \{n \in \mathbb{N} \mid n \neq 2$ *und* $n \neq 3$ *und* $n \neq 5$ *und* $n \neq 7\} = \{1, 4, 6, 8, 9, 10, 11, 12, \dots\}$*.*

Als Potenzmenge der Menge A ergibt sich die folgende Menge von Mengen von natürlichen Zahlen $\mathcal{P}(A) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{7\}, \{2, 3\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \{2, 3, 5\}, \{2, 3, 7\}, \{2, 5, 7\}, \{3, 5, 7\}, \{2, 3, 5, 7\}\}$*.*

Offensichtlich ist die Menge $\{0, 2, 4, 6, 8, \dots\}$ *der geraden natürlichen Zahlen und die Menge* $\{1, 3, 5, 7, 9, \dots\}$ *der ungeraden natürlichen Zahlen disjunkt.*

A.1.4. Gesetze für Mengenoperationen

Für die klassischen Mengenoperationen gelten die folgenden Beziehungen:

$A \cap B$	$=$	$B \cap A$	Kommutativgesetz für den Schnitt
$A \cup B$	$=$	$B \cup A$	Kommutativgesetz für die Vereinigung
$A \cap (B \cap C)$	$=$	$(A \cap B) \cap C$	Assoziativgesetz für den Schnitt
$A \cup (B \cup C)$	$=$	$(A \cup B) \cup C$	Assoziativgesetz für die Vereinigung
$A \cap (B \cup C)$	$=$	$(A \cap B) \cup (A \cap C)$	Distributivgesetz
$A \cup (B \cap C)$	$=$	$(A \cup B) \cap (A \cup C)$	Distributivgesetz
$A \cap A$	$=$	A	Duplizitätsgesetz für den Schnitt
$A \cup A$	$=$	A	Duplizitätsgesetz für die Vereinigung
$A \cap (A \cup B)$	$=$	A	Absorptionsgesetz
$A \cup (A \cap B)$	$=$	A	Absorptionsgesetz
$\overline{A \cap B}$	$=$	$(\overline{A} \cup \overline{B})$	de-Morgansche Regel
$\overline{A \cup B}$	$=$	$(\overline{A} \cap \overline{B})$	de-Morgansche Regel
$\overline{\overline{A}}$	$=$	A	Gesetz des doppelten Komplements

Die „de-Morganschen Regeln“ wurden nach dem englischen Mathematiker Augustus De Morgan[21] benannt.

Als Abkürzung schreibt man statt $X_1 \cup X_2 \cup \cdots \cup X_n$ (bzw. $X_1 \cap X_2 \cap \cdots \cap X_n$) einfach $\bigcup_{i=1}^{n} X_i$ (bzw. $\bigcap_{i=1}^{n} X_i$). Möchte man alle Mengen X_i mit $i \in \mathbb{N}$ schneiden (bzw. vereinigen), so schreibt man kurz $\bigcap_{i \in \mathbb{N}} X_i$ (bzw. $\bigcup_{i \in \mathbb{N}} X_i$).

Oft benötigt man eine Verknüpfung von zwei Mengen, eine solche Verknüpfung wird allgemein wie folgt definiert:

Definition 78 („Verknüpfung von Mengen“): *Seien A und B zwei Mengen und „$\odot$“ eine beliebige Verknüpfung zwischen den Elementen dieser Mengen, dann definieren wir*

$$A \odot B =_{\text{def}} \{a \odot b \mid a \in A \text{ und } b \in B\}.$$

Beispiel 79: *Die Menge $3\mathbb{Z} = \{0, \pm 3, \pm 6, \pm 9, \dots\}$ enthält alle Vielfachen[22] von 3, damit ist $3\mathbb{Z} + \{1\} = \{1, 4, -2, 7, -5, 10, -8, \dots\}$. Die Menge $3\mathbb{Z} + \{1\}$ schreibt man kurz oft auch als $3\mathbb{Z} + 1$, wenn klar ist, was mit dieser Abkürzung gemeint ist.*

A.1.5. Tupel (Vektoren) und das Kreuzprodukt

Seien $A, A_1, \dots, A_n$ im folgenden Mengen, dann bezeichnet

- $(a_1, \dots, a_n) =_{\text{def}}$ die Elemente $a_1, \dots, a_n$ in genau dieser festgelegten *Reihenfolge* und z.B. gilt damit $(3, 2) \neq (2, 3)$. Wir sprechen von einem n-Tupel.
- $A_1 \times A_2 \times \cdots \times A_n =_{\text{def}} \{(a_1, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$ (Kreuzprodukt der Mengen $A_1, A_2, \dots, A_n$),
- $A^n =_{\text{def}} \underbrace{A \times A \times \cdots \times A}_{n\text{-mal}}$ (n-faches Kreuzprodukt der Menge A) und
- speziell gilt $A^1 = \{(a) \mid a \in A\}$.

[21] *1806 in Madurai, Tamil Nadu, Indien - †1871 in London, England

[22] Eigentlich müsste man statt $3\mathbb{Z}$ die Notation $\{3\}\mathbb{Z}$ verwenden. Dies ist allerdings unüblich.

Wir nennen 2-Tupel auch *Paare*, 3-Tupel auch *Tripel*, 4-Tupel auch *Quadrupel* und 5-Tupel *Quintupel*. Bei n-Tupeln ist, im Gegensatz zu Mengen, eine Reihenfolge vorgegeben, d.h. es gilt z.B. immer $\{a, b\} = \{b, a\}$, aber im Allgemeinen $(a, b) \neq (b, a)$.

Beispiel 80: *Sei $A = \{1, 2, 3\}$ und $B = \{a, b, c\}$, dann bezeichnet das Kreuzprodukt von A und B die Menge von Paaren $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$.*

A.1.6. Die Anzahl von Elementen in Mengen

Sei A eine Menge, die endlich viele Elemente[23] enthält, dann ist

$$\#A =_{\text{def}} \text{Anzahl der Elemente in der Menge } A.$$

Beispielsweise ist $\#\{4, 7, 9\} = 3$. Mit dieser Definition gilt

- $\#(A^n) = (\#A)^n$,
- $\#\mathcal{P}(A) = 2^{\#A}$,
- $\#A + \#B = \#(A \cup B) + \#(A \cap B)$ und
- $\#A = \#(A \setminus B) + \#(A \cap B)$.

A.2. Relationen und Funktionen

A.2.1. Eigenschaften von Relationen

Seien $A_1, \ldots, A_n$ beliebige Mengen, dann ist R eine *n-stellige Relation* gdw. $R \subseteq A_1 \times A_2 \times \cdots \times A_n$. Eine zweistellige Relation nennt man auch *binäre Relation*. Oft werden auch Relationen $R \subseteq A^n$ betrachtet, diese bezeichnet man dann als n-stellige Relation über der Menge A.

Definition 81: *Sei R eine zweistellige Relation über A, dann ist R*

- reflexiv *gdw. $(a, a) \in R$ für alle $a \in A$,*
- symmetrisch *gdw. aus $(a, b) \in R$ folgt $(b, a) \in R$,*
- antisymmetrisch *gdw. aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$,*
- transitiv *gdw. aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ und*
- linear *gdw. es gilt immer $(a, b) \in R$ oder $(b, a) \in R$.*

Definition 82: *Sei R eine zweistellige Relation, dann*

- *heißt R* Halbordnung *gdw. R ist reflexiv, antisymmetrisch und transitiv,*
- Ordnung *gdw. R ist eine lineare Halbordnung und*
- Äquivalenzrelation *gdw. R reflexiv, transitiv und symmetrisch ist.*

Beispiel 83: *Die Teilmengenrelation „$\subseteq$“ auf allen Teilmengen von $\mathbb{Z}$ ist eine Halbordnung, aber keine Ordnung.*

[23] Solche Mengen werden als *endliche Mengen* bezeichnet.

Beispiel 84: *Wir schreiben $a \equiv b \bmod n$, falls es eine ganze Zahl q gibt, für die $a - b = qn$ gilt. Für $n \geq 2$ ist die Relation $R_n(a,b) =_{\text{def}} \{(a,b) \mid a \equiv b \bmod n\} \subseteq \mathbb{Z}^2$ eine Äquivalenzrelation.*

A.2.2. Eigenschaften von Funktionen

Seien A und B beliebige Mengen. f ist eine *Funktion* von A nach B (Schreibweise: $f\colon A \to B$) gdw. $f \subseteq A \times B$ und für jedes $a \in A$ gibt es *höchstens* ein $b \in B$ mit $(a,b) \in f$. Ist also $(a,b) \in f$, so schreibt man $f(a) = b$. Ebenfalls gebräuchlich ist die Notation $a \mapsto b$.

Bemerkung 85: *Unsere Definition von Funktion umfasst auch mehrstellige Funktionen. Seien C und B Mengen und $A = C^n$ das n-fache Kreuzprodukt von C. Die Funktion $f\colon A \to B$ ist dann eine n-stellige Funktion, denn sie bildet n-Tupel aus C^n auf Elemente aus B ab.*

Definition 86: *Sei f eine n-stellige Funktion. Möchte man die Funktion f benutzen, aber keine Namen für die Argumente vergeben, so schreibt man auch*

$$f(\underbrace{\cdot, \cdot, \ldots, \cdot}_{n\text{-mal}})$$

Ist also der Namen des Arguments einer einstelligen Funktion $g(x)$ für eine Betrachtung unwichtig, so kann man $g(\cdot)$ schreiben, um anzudeuten, dass g einstellig ist, ohne dies weiter zu erwähnen.

Definition 87: *Sei nun $R \subseteq A_1 \times A_2 \times \cdots \times A_n$ eine n-stellige Relation, dann definieren wir eine Funktion $P_R^n\colon A_1 \times A_2 \times \cdots \times A_n \to \{0,1\}$ wie folgt:*

$$P_R^n(x_1, \ldots, x_n) =_{\text{def}} \begin{cases} 1, & \text{falls } (x_1, \ldots, x_n) \in R \\ 0, & \text{sonst} \end{cases}$$

Eine solche n-stellige Funktion, die „anzeigt“, ob ein Element aus $A_1 \times A_2 \times \cdots \times A_n$ entweder zu R gehört oder nicht, nennt man (n-stelliges) Prädikat.

Beispiel 88: *Sei $\mathbb{P} =_{\text{def}} \{n \in \mathbb{N} \mid n \text{ ist Primzahl}\}$, dann ist $\mathbb{P}$ eine 1-stellige Relation über den natürlichen Zahlen. Das Prädikat $P_{\mathbb{P}}^1(n)$ liefert für eine natürliche Zahl n genau dann 1, wenn n eine Primzahl ist.*

Ist für ein Prädikat P_R^n sowohl die Relation R als auch die Stelligkeit n aus dem Kontext klar, dann schreibt man auch kurz P oder verwendet das Relationensymbol R als Notation für das Prädikat P_R^n.

Nun legen wir zwei spezielle Funktionen fest, die oft sehr hilfreich sind:

Definition 89: *Sei $\alpha \in \mathbb{R}$ eine beliebige reelle Zahl, dann gilt*

- $\lceil \alpha \rceil =_{\text{def}}$ *die kleinste ganze Zahl, die größer oder gleich α ist ($\triangleq$ „Aufrunden“)*
- $\lfloor \alpha \rfloor =_{\text{def}}$ *die größte ganze Zahl, die kleiner oder gleich α ist ($\triangleq$ „Abrunden“)*

Definition 90: *Für eine beliebige Funktion f legen wir fest:*

- *Der* Definitionsbereich *von f ist $D_f =_{\text{def}} \{a \mid$ es gibt ein b mit $f(a) = b\}$.*
- *Der* Wertebereich *von f ist $W_f =_{\text{def}} \{b \mid$ es gibt ein a mit $f(a) = b\}$.*
- *Die Funktion $f\colon A \to B$ ist* total *gdw. $D_f = A$.*

- *Die Funktion $f\colon A \to B$ heißt* surjektiv *gdw. $W_f = B$.*
- *Die Funktion f heißt* injektiv *(oder eineindeutig[24]) gdw. immer wenn $f(a_1) = f(a_2)$ gilt auch $a_1 = a_2$.*
- *Die Funktion f heißt* bijektiv *gdw. f ist injektiv und surjektiv.*

Mit Hilfe der Kontraposition (siehe Abschnitt C.1.1) kann man für die Injektivität alternativ auch zeigen, dass immer wenn $a_1 \neq a_2$, dann muss auch $f(a_1) \neq f(a_2)$ gelten.

Beispiel 91: *Sei die Funktion $f\colon \mathbb{N} \to \mathbb{Z}$ durch $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$ gegeben. Die Funktion f ist surjektiv, denn $f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, \ldots$, d.h. die ungeraden natürlichen Zahlen werden auf die negativen ganzen Zahlen abgebildet, die geraden Zahlen aus $\mathbb{N}$ werden auf die positiven ganzen Zahlen abgebildet und deshalb ist $W_f = \mathbb{Z}$.*

Weiterhin ist f auch injektiv, denn aus[25] $(-1)^{a_1} \lceil \frac{a_1}{2} \rceil = (-1)^{a_2} \lceil \frac{a_2}{2} \rceil$ folgt, dass entweder a_1 und a_2 gerade oder a_1 und a_2 ungerade, denn sonst würden auf der linken und rechten Seite der Gleichung unterschiedliche Vorzeichen auftreten. Ist a_1 gerade und a_2 gerade, dann gilt $\lceil \frac{a_1}{2} \rceil = \lceil \frac{a_2}{2} \rceil$ und auch $a_1 = a_2$. Sind a_1 und a_2 ungerade, dann gilt $-\lceil \frac{a_1}{2} \rceil = -\lceil \frac{a_2}{2} \rceil$, woraus auch folgt, dass $a_1 = a_2$. Damit ist die Funktion f bijektiv. Weiterhin ist f auch total, d.h. $D_f = \mathbb{N}$.

Definition 92: *Unter einem n-stelligen* Operator *f (auf der Menge Y) versteht man in der Mathematik eine Funktion der Form $f\colon Y^n \to Y$. Einfache Beispiele für zweistellige Operatoren sind der Additions- oder Multiplikationsoperator.*

A.2.3. Hüllenoperatoren

Definition 93: *Sei X eine Menge. Ein einstelliger Operator $\Psi\colon \mathcal{P}(X) \to \mathcal{P}(X)$ heißt* Hüllenoperator*, wenn er die folgenden drei Eigenschaften erfüllt:*

Einbettung: *für alle $A \in \mathcal{P}(X)$ gilt $A \subseteq \Psi(A)$*

Monotonie: *für alle $A, B \in \mathcal{P}(X)$ mit $A \subseteq B$ folgt $\Psi(A) \subseteq \Psi(B)$*

Abgeschlossenheit: *für alle $A \in \mathcal{P}(X)$ gilt $\Psi(\Psi(A)) = \Psi(A)$*

Aufgrund der Monotonieeigenschaft eines Hüllenoperators kann man bei der Abgeschlossenheit die Eigenschaft $\Psi(\Psi(A)) = \Psi(A)$ auch durch $\Psi(\Psi(A)) \subseteq \Psi(A)$ ersetzen. In der Informatik spielen Hüllenoperatoren eine große Rolle. Gute Beispiele hierfür sind z.B. die *transitive Hülle* (vgl. Computergraphik), die Kleene-Hülle (vgl. Formale Sprachen) oder der Abschluss einer Komplexitätsklasse unter Schnitt oder Vereinigung.

A.2.4. Permutationen

Sei S eine beliebige endliche Menge, dann heißt eine bijektive Funktion π der Form $\pi\colon S \to S$ *Permutation*. Das bedeutet, dass die Funktion π Elemente aus S wieder auf Elemente aus S abbildet, wobei für jedes $b \in S$ ein $a \in S$ mit $f(a) = b$ existiert (Surjektivität) und falls $f(a_1) = f(a_2)$ gilt, dann ist $a_1 = a_2$ (Injektivität).

Bemerkung 94: *Man kann den Permutationsbegriff auch auf unendliche Mengen erweitern, aber besonders häufig werden in der Informatik* Permutationen von endlichen Mengen *benötigt. Aus diesem Grund sollen hier nur endliche Mengen S betrachtet werden.*

[24] Achtung: Dieser Begriff wird manchmal unterschiedlich, je nach Autor, in den Bedeutungen „bijektiv“ oder „injektiv“ verwendet.

[25] Für die Definition der Funktion $\lceil \cdot \rceil$ siehe Definition 89.

Sei nun $S = \{1, \ldots, n\}$ (eine endliche Menge) und $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation. Permutationen dieser Art kann man sehr anschaulich mit Hilfe einer Matrix aufschreiben:

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

Durch diese Notation wird klar, dass das Element 1 der Menge S durch das Element $\pi(1)$ ersetzt wird, das Element 2 wird mit $\pi(2)$ vertauscht und allgemein das Element i durch $\pi(i)$ für $1 \leq i \leq n$. In der zweiten Zeile dieser Matrixnotation findet sich also *jedes* (Surjektivität) Element der Menge S genau *einmal* (Injektivität).

Beispiel 95: *Sei $S = \{1, \ldots, 3\}$ eine Menge mit drei Elementen. Dann gibt es, wie man ausprobieren kann, genau 6 Permutationen von S:*

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad \pi_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \quad \pi_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$
$$\pi_4 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad \pi_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \quad \pi_6 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Satz 96: *Sei S eine endliche Menge mit $n = |S|$, dann gibt es genau $n!$ (Fakultät) verschiedene Permutationen von S.*

Beweis: Jede Permutation π der Menge S von n Elementen kann als Matrix der Form

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

aufgeschrieben werden. Damit ergibt sich die Anzahl der Permutationen von S durch die Anzahl der verschiedenen zweiten Zeilen solcher Matrizen. In jeder solchen Zeile muss jedes der n Elemente von S genau einmal vorkommen, da π eine bijektive Abbildung ist, d.h. wir haben für die erste Position der zweiten Zeile der Matrixdarstellung genau n verschiedene Möglichkeiten, für die zweite Position noch $n-1$ und für die dritte noch $n-2$. Für die n-te Position bleibt nur noch 1 mögliches Element aus S übrig[26]. Zusammengenommen haben wir also $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 = n!$ verschiedene mögliche Permutationen der Menge S. #

A.3. Summen und Produkte

A.3.1. Summen

Zur abkürzenden Schreibweise verwendet man für Summen das Summenzeichen $\sum$. Dabei ist

$$\sum_{i=1}^{n} a_i =_{\text{def}} a_1 + a_2 + \cdots + a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\sum_{i=1}^{n} a_i = n \cdot a$ (Summe gleicher Summanden).
- $\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten einer Summe).

[26] Dies kann man sich auch als die Anzahl der verschiedenen Möglichkeiten vorstellen, die bestehen, wenn man aus einer Urne mit n numerierten Kugeln alle Kugeln *ohne* Zurücklegen nacheinander zieht.

- $\sum_{i=1}^{n}(a_i + b_i + c_i + \dots) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i + \sum_{i=1}^{n} c_i + \dots$ (Addition von Summen).
- $\sum_{i=1}^{n} a_i = \sum_{i=l}^{n+l-1} a_{i-l+1}$ und $\sum_{i=l}^{n} a_i = \sum_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Summen).
- $\sum_{i=1}^{n}\sum_{j=1}^{m} a_{i,j} = \sum_{j=1}^{m}\sum_{i=1}^{n} a_{i,j}$ (Vertauschen der Summationsfolge).

Manchmal verwendet man keine Laufindizes an ein Summenzeichen, sondern man beschreibt (durch ein Prädikat) welche Zahlen aufsummiert werden sollen. So kann man eine Funktion definieren, die die Summe aller Teiler einer natürlichen Zahl liefert:

$$\sigma(n) =_{\text{def}} \sum_{\substack{t \le n \\ t \text{ teilt } n}} t$$

A.3.2. Produkte

Zur abkürzenden Schreibweise verwendet man für Produkte das Produktzeichen $\prod$. Dabei ist

$$\prod_{i=1}^{n} a_i =_{\text{def}} a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \le i \le n$, dann gilt $\prod_{i=1}^{n} a_i = a^n$ (Produkt gleicher Faktoren).
- $\prod_{i=1}^{n}(ca_i) = c^n \prod_{i=1}^{n} a_i$ (Vorziehen von konstanten Faktoren)
- $\prod_{i=1}^{n} a_i = \prod_{i=1}^{m} a_i \cdot \prod_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten in Teilprodukte).
- $\prod_{i=1}^{n}(a_i \cdot b_i \cdot c_i \cdot \ldots) = \prod_{i=1}^{n} a_i \cdot \prod_{i=1}^{n} b_i \cdot \prod_{i=1}^{n} c_i \cdot \ldots$ (Das Produkt von Produkten).
- $\prod_{i=1}^{n} a_i = \prod_{i=l}^{n+l-1} a_{i-l+1}$ und $\prod_{i=l}^{n} a_i = \prod_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Produkten).
- $\prod_{i=1}^{n}\prod_{j=1}^{m} a_{i,j} = \prod_{j=1}^{m}\prod_{i=1}^{n} a_{i,j}$ (Vertauschen der Reihenfolge bei Doppelprodukten).

Ähnlich wie bei Summen kann man bei Produkten auch ohne Laufindex arbeiten. So ist z.B. die *Eulersche ϕ-Funktion* wie folgt definiert:

$$\phi(n) =_{\text{def}} n \prod_{p|n}(1 - \frac{1}{p})$$

In das Produkt gehen also alle Primteiler p von n mit dem Faktor $1 - 1/p$ ein.

Oft werden Summen- oder Produktsymbole verwendet bei denen der Startindex größer als der Stopindex ist. Solche Summen bzw. Produkte sind „leer“, d.h. es wird nichts summiert bzw. multipliziert. Sind dagegen Start- und Endindex gleich, so tritt nur genau ein Wert auf, d.h. das Summen- bzw. Produktsymbol hat in diesem Fall keine Auswirkung. Es ergeben sich also die folgenden Rechenregeln:

- Seien $n, m \in \mathbb{Z}$ und $n < m$, dann

$$\sum_{i=m}^{n} a_i = 0 \text{ und } \prod_{i=m}^{n} a_i = 1$$

- Sei $n \in \mathbb{Z}$, dann

$$\sum_{i=n}^{n} a_i = a_n = \prod_{i=n}^{n} a_i$$

Beispiel 97: *Die folgende Identität wird Euler zugeschrieben. Erstaunlicherweise kann man damit eine Summe von Kehrwerten von Potenzen mit einem Produkt von Primzahlen in Verbindung bringen.*

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p} \frac{1}{1 - \frac{1}{p^s}}$$

Diese Identiät kann man Die Summe auf der linken Seite ist auch als Riemannsche Zetafuntion $\zeta(s)$ *bekannt. Erstaunlicherweise gilt dann der folgende Zusammenhang*

$$\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \prod_{p} \frac{1}{1 - \frac{1}{p^2}} = \frac{\pi^2}{6},$$

denn durch diese Gleichung wird die Menge der Primzahlen in Beziehung zur Kreiszahl π gebracht.

A.4. Logarithmieren, Potenzieren und Radizieren

Die Schreibweise a^b ist eine Abkürzung für

$$a^b =_{\text{def}} \underbrace{a \cdot a \cdot \ldots \cdot a}_{b-\text{mal}}$$

und wird als *Potenzierung* bezeichnet. Dabei ist a die *Basis*, b der *Exponent* und a^b die b-te *Potenz* von a. Seien nun $r, s, t \in \mathbb{R}$ und $r, t \geq 0$ durch die folgende Gleichung verbunden:

$$r^s = t.$$

Dann lässt sich diese Gleichung wie folgt umstellen und es gelten die folgenden Rechenregeln:

Logarithmieren	Potenzieren	Radizieren
$\mathbf{s = \log_r t}$	$\mathbf{t = r^s}$	$\mathbf{r = \sqrt[s]{t}}$
i) $\log_r(\frac{u}{v}) = \log_r u - \log_r v$	i) $r^u \cdot r^v = r^{u+v}$	i) $\sqrt[s]{u} \cdot \sqrt[s]{v} = \sqrt[s]{u \cdot v}$
ii) $\log_r(u \cdot v) = \log_r u + \log_r v$	ii) $\frac{r^u}{r^v} = r^{u-v}$	ii) $\frac{\sqrt[s]{u}}{\sqrt[s]{v}} = \sqrt[s]{\left(\frac{u}{v}\right)}$
iii) $\log_r(t^u) = u \cdot \log_r t$	iii) $u^s \cdot v^s = (u \cdot v)^s$	iii) $\sqrt[u]{\sqrt[v]{t}} = \sqrt[u \cdot v]{t}$
iv) $\log_r(\sqrt[u]{t}) = \frac{1}{u} \cdot \log_r t$	iv) $\frac{u^s}{v^s} = \left(\frac{u}{v}\right)^s$	
v) $\frac{\log_r t}{\log_r u} = \log_u t$ (Basiswechsel)	v) $(r^u)^v = r^{u \cdot v}$	

Zusätzlich gilt: Wenn $r > 1$, dann ist $s_1 < s_2$ gdw. $r^{s_1} < r^{s_2}$ (Monotonie).

Da $\sqrt[s]{t} = t^{\left(\frac{1}{s}\right)}$ gilt, können die Gesetze für das Radizieren leicht aus den Potenzierungsgesetzen abgeleitet werden. Weiterhin legen wir spezielle Schreibweisen für die Logarithmen zur Basis 10, e (Eulersche Zahl) und 2 fest: $\lg t =_{\text{def}} \log_{10} t$, $\ln t =_{\text{def}} \log_e t$ und $\text{lb}\, t =_{\text{def}} \log_2 t$.

A.5. Gebräuchliche griechische Buchstaben

In der Informatik, Mathematik und Physik ist es üblich, griechische Buchstaben zu verwenden. Ein Grund hierfür ist, dass es so möglich wird mit einer größeren Anzahl von Unbekannten arbeiten zu können, ohne unübersichtliche und oft unhandliche Indizes benutzen zu müssen.

Kleinbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
α	Alpha	β	Beta	γ	Gamma
δ	Delta	ϕ	Phi	φ	Phi
ξ	Xi	ζ	Zeta	ϵ	Epsilon
θ	Theta	λ	Lambda	π	Pi
σ	Sigma	η	Eta	μ	Mu

Großbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
Γ	Gamma	Δ	Delta	Φ	Phi
Ξ	Xi	Θ	Theta	Λ	Lambda
Π	Pi	Σ	Sigma	Ψ	Psi
Ω	Omega				

B. Einige (wenige) Grundlagen der elementaren Logik

Aussagen sind entweder *wahr* ($\triangleq 1$) oder *falsch* ($\triangleq 0$). So sind die Aussagen

„Wiesbaden liegt am Mittelmeer“ und „$1 = 7$“

sicherlich falsch, wogegen die Aussagen

„Wiesbaden liegt in Hessen“ und „$11 = 11$“

sicherlich wahr sind. Aussagen werden meist durch *Aussagenvariablen* formalisiert, die nur die Werte 0 oder 1 annehmen können. Oft verwendet man auch eine oder mehrere Unbekannte, um eine Aussage zu parametrisieren. So könnte „$P(x)$“ etwa für „Wiesbaden liegt im Bundesland x“ stehen, d.h. „P(Hessen)“ wäre wahr, wogegen „P(Bayern)“ eine falsche Aussage ist. Solche Konstrukte mit Parameter nennt man auch *Prädikat* oder *Aussageformen.*

Um die Verknüpfung von Aussagen auch formal aufschreiben zu können, werden die folgenden logischen Operatoren verwendet

Symbol	umgangssprachlicher Name	Name in der Logik
$\wedge$	und	Konjunktion
$\vee$	oder	Disjunktion / Alternative
$\neg$	nicht	Negation
$\rightarrow$	folgt	Implikation
$\leftrightarrow$	genau dann wenn (*gdw.*)	Äquivalenz

Zusätzlich werden noch die Quantoren $\exists$ („es existiert“) und $\forall$ („für alle“) verwendet, die z.B. wie folgt gebraucht werden können

$\forall x\colon P(x)$ bedeutet „Für alle x gilt die Aussage $P(x)$“.

$\exists x\colon P(x)$ bedeutet „Es existiert ein x, für das die Aussage $P(x)$ gilt“.

Üblicherweise läßt man sogar den Doppelpunkt weg und schreibt statt $\forall x\colon P(x)$ vereinfachend $\forall x P(x)$. Die Aussageform $P(x)$ wird auch als Prädikat (siehe Definition 87 auf Seite 99) bezeichnet, weshalb man von *Prädikatenlogik* spricht.

Beispiel 98: *Die Aussage „Jede gerade natürliche Zahl kann als Produkt von* 2 *und einer anderen natürlichen Zahl geschrieben werden“ lässt sich dann wie folgt schreiben*

$$\forall n \in \mathbb{N}\colon ((n \textit{ ist gerade}) \rightarrow (\exists m \in \mathbb{N}\colon n = 2 \cdot m))$$

Die folgende logische Formel wird wahr gdw. n eine ungerade natürliche Zahl ist.

$$\exists m \in \mathbb{N}\colon (n = 2 \cdot m + 1)$$

Für die logischen Konnektoren sind die folgenden Wahrheitswertetafeln festgelegt:

p	$\neg p$
0	1
1	0

und

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Jetzt kann man Aussagen auch etwas komplexer verknüpfen:

Beispiel 99: *Nun wird der $\wedge$-Operator verwendet werden. Dazu soll die Aussage „Für alle natürlichen Zahlen n und m gilt, wenn n kleiner gleich m und m kleiner gleich n gilt, dann ist m gleich n“*

$$\forall n, m \in \mathbb{N}\,(((n \leq m) \wedge (m \leq n)) \rightarrow (n = m))$$

Oft benutzt man noch den negierten Quantor $\nexists$ („es existiert kein“).

Cubum autem in duos cubos, aut quadrato-quadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Beispiel 100 („Großer Satz von Fermat“): *Die Richtigkeit dieser Aussage konnte erst* 1994 *nach mehr als* 350 *Jahren von Andrew Wiles und Richard Taylor gezeigt werden:*

$$\forall n \in \mathbb{N}\,\nexists a, b, c \in \mathbb{N}\,(((n > 2) \wedge (a \cdot b \cdot c \neq 0)) \rightarrow a^n + b^n = c^n)$$

Für den Fall $n = 2$ hat die Gleichung $a^n + b^n = c^n$ unendlich viele ganzzahlige Lösungen (die so genannten Pythagoräischen Zahlentripel) wie z.B. $3^2 + 4^2 = 5^2$. Diese sind seit mehr als 3500 *Jahren bekannt und haben z.B. geholfen die Cheops-Pyramide zu bauen.*

C. Einige formale Grundlagen von Beweistechniken

Praktisch arbeitende Informatiker glauben oft völlig ohne (formale) Beweistechniken auskommen zu können. Dabei meinen sie sogar, dass formale Beweise keinerlei Berechtigung in der Praxis der Informatik haben und bezeichnen solches Wissen als „in der Praxis irrelevantes Zeug, das nur von und für seltsame Wissenschaftler erfunden wurde“. Studenten in den ersten Semestern unterstellen sogar oft, dass mathematische Grundlagen und Beweistechniken nur als „Filter“ dienen, um die Anzahl der Studenten zu reduzieren. Oft stellen sich beide Gruppen auf den Standpunkt, dass die Korrektheit von Programmen und Algorithmen durch „Lassen wir es doch mal laufen und probieren es aus!“ ($\triangleq$ Testen) belegt werden könne[27]. Diese Einstellung zeigt sich oft auch darin, dass Programme mit Hilfe einer IDE schnell „testweise“ übersetzt werden, in der Hoffnung oder (wesentlich schlimmer) in der Überzeugung, dass jedes übersetzbare Programm ($\triangleq$ syntaktisch korrekt) automatisch auch semantisch korrekt ist. Das diese Einstellung völlig falsch ist zeigen Myriaden in der Praxis auftretende Softwarefehler eindrücklich.

Theoretiker, die sich mit den Grundlagen der Informatik beschäftigen, vertreten oft den Standpunkt, dass die Korrektheit *jedes* Programms rigoros *bewiesen* werden muss. Wahrscheinlich ist die Position zwischen diesen beiden Extremen richtig, denn zum einen ist der formale Beweis von (großen) Programmen oft nicht praktikabel oder aufgrund der enormen Komplexität nicht möglich und zum anderen kann das Testen mit einer (relativ kleinen) Menge von Eingaben sicherlich nicht belegen, dass ein Programm vollständig den Spezifikationen entspricht. Im praktischen Einsatz ist es dann oft mit Eingaben konfrontiert, die zu einer fehlerhaften Reaktion führen oder es sogar abstürzen[28] lassen. Bei einfacher Anwendersoftware sind solche Fehler ärgerlich, aber oft zu verschmerzen. Bei sicherheitskritischer Software (z.B. bei der Regelung von Atomkraftwerken, Airbags und Bremssystemen in Autos, in der Medizintechnik, bei Finanztransaktionssystemen oder bei der Steuerung von Raumsonden) gefährden solche Fehler menschliches Leben oder führen zu extrem hohen finanziellen Verlusten und müssen deswegen unbedingt vermieden werden.

Für den Praktiker bringen Kenntnisse über formale Beweise aber noch andere Vorteile. Viele Beweise beschreiben direkt den zur Lösung benötigten Algorithmus, d.h. eigentlich wird die Richtigkeit einer Aussage durch die (implizite) Angabe eines Algorithmus gezeigt. Aber es gibt noch einen anderen Vorteil. Ist der umzusetzende Algorithmus komplex (z.B. aufgrund einer komplizierten Schleifenstruktur oder einer verschachtelten Rekursion), so ist es unwahrscheinlich, eine korrekte Implementation an den Kunden liefern zu können, ohne die Hintergründe ($\triangleq$ Beweis) verstanden zu haben. All dies zeigt, dass auch ein praktischer Informatiker Einblicke in Beweistechniken haben sollte. Interessanterweise zeigt die persönliche Erfahrung im praktischen Umfeld auch, dass solches (theoretisches) Wissen über die Hintergründe oft zu klarer strukturierten und effizienteren Programmen führt.

Aus diesen Gründen sollen in den folgenden Abschnitten einige grundlegende Beweistechniken mit Hilfe von Beispielen (unvollständig) kurz vorgestellt werden.

C.1. Direkte Beweise

Um einen *direkten Beweis* zu führen, müssen wir, beginnend von einer initialen Aussage ($\triangleq$ Hypothese), durch Angabe einer Folge von (richtigen) Zwischenschritten zu der zu beweisenden Aussage ($\triangleq$ Folgerung) gelangen. Jeder Zwischenschritt ist dabei entweder unmittelbar klar oder muss wieder durch einen weiteren (kleinen) Beweis belegt werden. Dabei müssen nicht alle

[27] Diese Bemerkung ist natürlich etwas polemisch, da *richtiges* Testen natürlich die Qualität einer Software verbessert. Weiterhin werden solche Tests systematisch durchgeführt und haben somit eine ganz andere Qualität als „herumprobieren“.

[28] Dies wird eindrucksvoll durch viele Softwarepakete und verbreitete Betriebssysteme im PC-Umfeld belegt.

Schritte völlig formal beschrieben werden, sondern es kommt darauf an, dass sich dem Leser die eigentliche Strategie erschließt.

Satz 101: *Sei* $n \in \mathbb{N}$. *Falls* $n \geq 4$, *dann ist* $2^n \geq n^2$.

Wir müssen also, in Abhängigkeit des Parameters n, die Richtigkeit dieser Aussage belegen. Einfaches Ausprobieren ergibt, dass $2^4 = 16 \geq 16 = 4^2$ und $2^5 = 32 \geq 25 = 5^2$, d.h. intuitiv scheint die Aussage richtig zu sein. Wir wollen die Richtigkeit der Aussage nun durch eine Reihe von (kleinen) Schritten belegen:
Beweis:

Wir haben schon gesehen, dass die Aussage für $n = 4$ und $n = 5$ richtig ist. Erhöhen wir n auf $n+1$, so verdoppelt sich der Wert der linken Seite der Ungleichung von 2^n auf $2 \cdot 2^n = 2^{n+1}$. Für die rechte Seite ergibt sich ein Verhältnis von $(\frac{n+1}{n})^2$. Je größer n wird, desto kleiner wird der Wert $\frac{n+1}{n}$, d.h. der maximale Wert ist bei $n = 4$ mit 1.25 erreicht. Wir wissen $1.25^2 = 1.5625$, d.h. immer wenn wir n um eins erhöhen, verdoppelt sich der Wert der linken Seite, wogegen sich der Wert der rechten Seite um maximal das 1.5625-fache erhöht. Damit muss die linke Seite der Ungleichung immer größer als die rechte Seite sein. #

Dieser Beweis war nur wenig formal, aber sehr ausführlich und wurde am Ende durch das Symbol „#“ markiert. Im Laufe der Zeit hat es sich eingebürgert, das Ende eines Beweises mit einem besonderen Marker abzuschließen. Besonders bekannt ist hier „qed“, eine Abkürzung für die lateinische Floskel „quod erat demonstrandum“, die mit „was zu beweisen war“ übersetzt werden kann. In neuerer Zeit werden statt „qed“ mit der gleichen Bedeutung meist die Symbole „□“ oder „#“ verwendet.

Nun stellt sich die Frage: „Wie formal und ausführlich muss ein Beweis sein?“ Diese Frage kann so einfach nicht beantwortet werden, denn das hängt u.a. davon ab, welche Lesergruppe durch den Beweis von der Richtigkeit einer Aussage überzeugt werden soll und wer den Beweis schreibt. Ein Beweis für ein Übungsblatt sollte auch auf Kleinigkeiten Rücksicht nehmen, wogegen ein solcher Stil für eine wissenschaftliche Zeitschrift vielleicht nicht angebracht wäre, da die potentielle Leserschaft über ganz andere Erfahrungen und viel mehr Hintergrundwissen verfügt.

Nun noch eine Bemerkung zum Thema „Formalismus“. Die menschliche Sprache ist unpräzise, mehrdeutig und Aussagen können oft auf verschiedene Weise interpretiert werden, wie das tägliche Zusammenleben der Menschen und die „Juristerei“ eindrucksvoll demonstriert. Diese Defizite sollen Formalismen[29] ausgleichen, d.h. die Antwort muss lauten: „So viele Formalismen wie notwendig und so wenige wie möglich!“. Durch Übung und Praxis lernt man die Balance zwischen diesen Anforderungen zu halten und es zeigt sich bald, dass „Geübte“ die formale Beschreibung sogar wesentlich leichter verstehen.

Oft kann man andere, schon bekannte, Aussagen dazu verwenden, die Richtigkeit einer neuen (evtl. kompliziert wirkenden) Aussage zu belegen.

Satz 102: *Sei* $n \in \mathbb{N}$ *die Summe von* 4 *Quadratzahlen, die größer als* 0 *sind, dann muss* $2^n \geq n^2$ *sein.*

Beweis: Die Menge der Quadratzahlen ist $\mathbb{S} = \{0, 1, 4, 9, 16, 25, 36, \dots\}$, d.h. 1 ist die kleinste Quadratzahl, die größer als 0 ist. Damit muss unsere Summe von 4 Quadratzahlen größer oder gleich als 4 sein. Die Aussage folgt direkt aus Satz 101. #

[29] In diesem Zusammenhang sind Programmiersprachen auch Formalismen, die eine präzise Beschreibung von Algorithmen erzwingen und die durch einen Compiler verarbeitet werden können.

C.1.1. Die Kontraposition

Mit Hilfe von direkten Beweisen haben wir Zusammenhänge der Form „Wenn Aussage H richtig ist, dann folgt daraus die Aussage C“ untersucht. Manchmal ist es schwierig, einen Beweis für eine solchen Zusammenhang zu finden. Völlig gleichwertig ist die Behauptung „Wenn die Aussage C falsch ist, dann ist die Aussage H falsch“ und oft ist eine solche Aussage leichter zu zeigen.

Die *Kontraposition* von Satz 101 ist also die folgende Aussage: „Wenn nicht $2^n \geq n^2$, dann gilt nicht $n \geq 4$.“. Das entspricht der Aussage: „Wenn $2^n < n^2$, dann gilt $n < 4$.“, was offensichtlich zu der ursprünglichen Aussage von Satz 101 gleichwertig ist.

Diese Technik ist oft besonders hilfreich, wenn man die Richtigkeit einer Aussage zeigen soll, die aus zwei Teilaussagen zusammengesetzt und die durch ein „genau dann wenn“[30] verknüpft sind. In diesem Fall sind zwei Teilbeweise zu führen, denn zum einen muss gezeigt werden, dass aus der ersten Aussage die zweite folgt und umgekehrt muss gezeigt werden, dass aus der zweiten Aussage die erste folgt.

Satz 103: *Eine natürliche Zahl n ist durch drei teilbar genau dann, wenn die Quersumme ihrer Dezimaldarstellung durch drei teilbar ist.*

Beweis: Für die Dezimaldarstellung von n gilt

$$n = \sum_{i=0}^{k} a_i \cdot 10^i, \text{wobei } a_i \in \{0, 1, \ldots, 9\} \text{ („Ziffern“) und } 0 \leq i \leq k.$$

Mit $\mathrm{QS}(n)$ wird die Quersumme von n bezeichnet, d.h. $\mathrm{QS}(n) = \sum_{i=0}^{k} a_i$. Mit Hilfe einer einfachen vollständigen Induktion kann man zeigen, dass für jedes $i \geq 0$ ein $b \in \mathbb{N}$ existiert, sodass $10^i = 9b + 1$. Damit gilt $n = \sum_{i=0}^{k} a_i \cdot 10^i = \sum_{i=0}^{k} a_i(9b_i + 1) = \mathrm{QS}(n) + 9\sum_{i=0}^{k} a_i b_i$, d.h. es existiert ein $c \in \mathbb{N}$, so dass $n = \mathrm{QS}(n) + 9c$.

„$\Rightarrow$“: Wenn n durch 3 teilbar ist, dann muss auch $\mathrm{QS}(n)+9c$ durch 3 teilbar sein. Da $9c$ sicherlich durch 3 teilbar ist, muss auch $\mathrm{QS}(n) = n - 9c$ durch 3 teilbar sein.

„$\Leftarrow$“: Dieser Fall soll durch Kontraposition gezeigt werden. Sei nun n nicht durch 3 teilbar, dann darf $\mathrm{QS}(n)$ nicht durch 3 teilbar sein, denn sonst wäre $n = 9c + \mathrm{QS}(n)$ durch 3 teilbar. #

C.2. Der Ringschluss

Oft findet man mehrere Aussagen, die zueinander äquivalent sind. Ein Beispiel dafür ist Satz 104. Um die Äquivalenz dieser Aussagen zu beweisen, müssten jeweils zwei „genau dann wenn“ Beziehungen untersucht werden, d.h. es werden vier Teilbeweise notwendig. Dies kann mit Hilfe eines so genannten *Ringschlusses* abgekürzt werden, denn es reicht zu zeigen, dass aus der ersten Aussage die zweite folgt, aus der zweiten Aussage die dritte und dass schließlich aus der dritten Aussage wieder die erste folgt. Im Beweis zu Satz 104 haben wir deshalb nur drei anstatt vier Teilbeweise zu führen, was zu einer Arbeitsersparnis führt. Diese Arbeitsersparnis wird um so größer, je mehr äquivalente Aussagen zu untersuchen sind. Dabei ist die Reihenfolge der Teilbeweise nicht wichtig, solange die einzelnen Teile zusammen einen Ring bilden.

Satz 104: *Seien A und B zwei beliebige Mengen, dann sind die folgenden drei Aussagen äquivalent:*

i) $A \subseteq B$

[30] Oft wird „genau dann wenn“ durch *gdw.* abgekürzt.

ii) $A \cup B = B$

iii) $A \cap B = A$

Beweis: Im folgenden soll ein Ringschluss verwendet werden, um die Äquivalenz der drei Aussagen zu zeigen:
„i) ⇒ ii)“: Da nach Voraussetzung $A \subseteq B$ ist, gilt für jedes Element $a \in A$ auch $a \in B$, d.h. in der Vereinigung $A \cup B$ sind alle Elemente nur aus B, was $A \cup B = B$ zeigt.

„ii) ⇒ iii)“: Wenn $A \cup B = B$ gilt, dann ergibt sich durch Einsetzen und mit den Regeln aus Abschnitt A.1.4 (Absorptionsgesetz) direkt $A \cap B = A \cap (A \cup B) = A$.

„iii) ⇒ i)“: Sei nun $A \cap B = A$, dann gibt es kein Element $a \in A$ für das $a \notin B$ gilt. Dies ist aber gleichwertig zu der Aussage $A \subseteq B$.

Damit hat sich ein Ring von Aussagen „i) ⇒ ii)“, „ii) ⇒ iii)“ und „iii) ⇒ i)“ gebildet, was die Äquivalenz aller Aussagen zeigt. #

C.3. Widerspruchsbeweise

Obwohl die Technik der Widerspruchsbeweise auf den ersten Blick sehr kompliziert erscheint, ist sie meist einfach anzuwenden, extrem mächtig und liefert oft sehr kurze Beweise. Angenommen wir sollen die Richtigkeit einer Aussage „aus der Hypothese H folgt C“ zeigen. Dazu beweisen wir, dass sich ein Widerspruch ergibt, wenn wir, von H und der Annahme, dass C falsch ist, ausgehen. Also war die Annahme falsch, und die Aussage C muss richtig sein.

Anschaulicher wird diese Beweistechnik durch folgendes Beispiel: Nehmen wir einmal an, dass Alice eine bürgerliche Frau ist und deshalb auch keine Krone trägt. Es ist klar, dass jede Königin eine Krone trägt. Wir sollen nun beweisen, dass Alice keine Königin ist. Dazu nehmen wir an, dass Alice eine Königin ist, d.h. Alice trägt eine Krone. Dies ist ein Widerspruch! Also war unsere Annahme falsch, und wir haben gezeigt, dass Alice keine Königin sein kann.

Der Beweis zu folgendem Satz verwendet diese Technik:

Satz 105: *Sei S eine endliche Untermenge einer unendlichen Menge U. Sei T das Komplement von S bzgl. U, dann ist T eine unendliche Menge.*

Beweis: Hier ist unsere Hypothese „S endlich, U unendlich und T Komplement von S bzgl. U“ und unsere Folgerung ist „T ist unendlich“. Wir nehmen also an, dass T eine endliche Menge ist. Da T das Komplement von S ist, gilt $S \cap T = \emptyset$, also ist $\#(S) + \#(T) = \#(S \cap T) + \#(S \cup T) = \#(S \cup T) = n$, wobei n eine Zahl aus $\mathbb{N}$ ist (siehe Abschnitt A.1.6). Damit ist $S \cup T = U$ eine endliche Menge. Dies ist ein Widerspruch zu unserer Hypothese! Also war die Annahme „T ist endlich“ falsch. #

C.4. Der Schubfachschluss

Der *Schubfachschluss* ist auch als *Dirichlets Taubenschlagprinzip* bekannt. Werden $n > k$ Tauben auf k Boxen verteilt, so gibt es mindestens eine Box in der sich wenigstens zwei Tauben aufhalten. Allgemeiner formuliert sagt das Taubenschlagprinzip, dass wenn n Objekte auf k Behälter aufgeteilt werden, dann gibt es mindestens eine Box die mindestens $\lceil \frac{n}{k} \rceil$ Objekte enthält.

Beispiel 106: *Auf einer Party unterhalten sich 8 Personen ($\triangleq$ Objekte), dann gibt es mindestens einen Wochentag ($\triangleq$ Box) an dem $\lceil \frac{8}{7} \rceil = 2$ Personen aus dieser Gruppe Geburtstag haben.*

C.5. Gegenbeispiele

Im wirklichen Leben wissen wir nicht, ob eine Aussage richtig oder falsch ist. Oft sind wir dann mit einer Aussage konfrontiert, die auf den ersten Blick richtig ist und sollen dazu ein Programm entwickeln. Wir müssen also entscheiden, ob diese Aussage wirklich richtig ist, denn sonst ist evtl. alle Arbeit umsonst und hat hohen Aufwand verursacht. In solchen Fällen kann man versuchen, ein einziges Beispiel dafür zu finden, dass die Aussage falsch ist, um so unnötige Arbeit zu sparen.

Wir zeigen, dass die folgenden Vermutungen falsch sind:

Vermutung 107: *Wenn $p \in \mathbb{N}$ eine Primzahl ist, dann ist p ungerade.*

Gegenbeispiel: Die natürliche Zahl 2 ist eine Primzahl und 2 ist gerade. #

Vermutung 108: *Es gibt keine Zahlen $a, b \in \mathbb{N}$, sodass $a \bmod b = b \bmod a$.*

Gegenbeispiel: Für $a = b = 2$ gilt $a \bmod b = b \bmod a = 0$. #

C.6. Induktionsbeweise und das Induktionsprinzip

Sei nun eine Menge von natürlichen Zahlen X mit den folgenden zwei Eigenschaften gegeben:

(IA) $0 \in X$
(IS) Ist eine beliebige natürliche Zahl n ein Element von X, so ist auch die Zahl $n + 1$ ein Element von X.

Man kann sich nun leicht überlegen, dass dann X alle natürlichen Zahlen enthält, d.h. es gilt $X = \mathbb{N}$. Mit Hilfe dieser Beobachtung konstruieren wir eine der nützlichsten Beweismethoden in der Informatik bzw. Mathematik: Das *Induktionsprinzip* bzw. die Methode des *Induktionsbeweises*. Die Idee funktioniert mit folgender Beobachtung:

> Angenommen man kann nachweisen, dass 0 die Eigenschaft E hat[31] (kurz: $E(0)$) und weiterhin, dass wenn n die Eigenschaft E hat, dann gilt auch $E(n+1)$. Ist dies der Fall, so muss jede natürliche Zahl die Eigenschaft E haben.

Im Folgenden wollen wir nachweisen, dass für jedes $n \in \mathbb{N}$ eine bestimmte Eigenschaft E gilt. Wir schreiben also abkürzend $E(n)$ für die Aussage „n besitzt die Eigenschaft E“, d.h. der Schreibweise $E(0)$ drücken wir also aus, dass die erste natürliche Zahl 0 die Eigenschaft E besitzt, dann erhalten wir die folgende Vorgehensweise:

Induktionsprinzip: Es gelten
(IA) $E(0)$
(IS) Für $n \geq 0$ gilt, wenn $E(n)$ korrekt ist, dann ist auch $E(n + 1)$ richtig.

Sind diese beiden Aussagen erfüllt, so hat jede natürliche Zahl die Eigenschaft E. Dabei ist **IA** die Abkürzung für *Induktionsanfang* und **IS** ist die Kurzform von *Induktionsschritt*. Die Voraussetzung ($\triangleq$ Hypothese) $E(n)$ ist korrekt für n und wird im Induktionsschritt als *Induktionsvoraussetzung* benutzt (kurz: **IV**). Hat man also den Induktionsanfang und den Induktionsschritt gezeigt, dann ist es anschaulich, dass jede natürliche Zahl die Eigenschaft E haben muss.

Es gibt verschiedene Versionen von Induktionsbeweisen. Die bekannteste Version ist die vollständige Induktion, bei der Aussagen über natürliche Zahlen gezeigt werden.

[31] Mit E wird also ein Prädikat oder Aussagenform bezeichnet (siehe Abschnitt A.1.2)

C.6.1. Die vollständige Induktion

Wie in Piratenfilmen üblich, seien Kanonenkugeln in einer Pyramide mit quadratischer Grundfläche gestapelt. Wir stellen uns die Frage, wieviele Kugeln (in Abhängigkeit von der Höhe) in einer solchen Pyramide gestapelt sind.

Satz 109: *Mit einer quadratische Pyramide aus Kanonenkugeln der Höhe $n \geq 1$ als Munition, können wir $\frac{n(n+1)(2n+1)}{6}$ Schüsse abgeben.*

Beweis: Einfacher formuliert: wir sollen zeigen, dass $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
(IA) Eine Pyramide der Höhe $n = 1$ enthält $\frac{1 \cdot 2 \cdot 3}{6} = 1$ Kugel, d.h. wir haben die Eigenschaft für $n = 1$ verifiziert.
(IV) Für $k \leq n$ gilt $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$.
(IS) Wir müssen nun zeigen, dass $\sum_{i=1}^{n+1} i^2 = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}$ gilt und dabei muss die Induktionsvoraussetzung $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ benutzt werden. Es ergeben sich die folgenden Schritte:

$$\begin{aligned}
\sum_{i=1}^{n+1} i^2 &= \sum_{i=1}^{n} i^2 + (n+1)^2 \\
&\overset{\text{(IV)}}{=} \frac{n(n+1)(2n+1)}{6} + (n^2+2n+1) \\
&= \frac{2n^3+3n^2+n}{6} + (n^2+2n+1) \\
&= \frac{2n^3+9n^2+13n+6}{6} \\
&= \frac{(n+1)(2n^2+7n+6)}{6} \quad (\star) \\
&= \frac{(n+1)(n+2)(2n+3)}{6} \quad (\star\star) \\
&= \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}
\end{aligned}$$

Die Zeile $\star$ (bzw. $\star\star$) ergibt sich, indem man $2n^3+9n^2+13n+6$ durch $n+1$ teilt (bzw. $2n^2+7n+6$ durch $n+2$). #

Das Induktionsprinzip kann man auch variieren. Dazu geht man davon aus, dass die Eigenschaft E für alle Zahlen $k \leq n$ erfüllt ist (Induktionsvoraussetzung). Obwohl dies auf den ersten Blick wesentlich komplizierter erscheint, vereinfacht dieser Ansatz oft den Beweis:

Verallgemeinertes Induktionsprinzip: Wenn die zwei Aussagen (also Induktionsanfang und Induktionsschritt)
(IA) $E(0)$
(IS) Wenn für alle $0 \leq k \leq n$ die Eigenschaft $E(k)$ gilt, dann ist auch $E(n+1)$ richtig,

gelten, dann haben alle natürliche Zahlen die Eigenschaft E. Damit ist das verallgemeinerte Induktionsprinzip eine Verallgemeinerung des weiter oben vorgestellten Induktionsprinzips, wie das folgende Beispiel veranschaulicht:

Satz 110: *Jede natürliche Zahl $n \geq 2$ läßt sich als Produkt von Primzahlen schreiben.*

Beweis: Das verallgemeinerte Induktionsprinzip wird wie folgt verwendet:
(IA) Offensichtlich ist 2 das Produkt von einer Primzahl.
(IV) Jede natürliche Zahl m mit $2 \leq m \leq n$ kann als Produkt von Primzahlen geschrieben werden.

(IS) Nun wird eine Fallunterscheidung durchgeführt:

i) Sei $n+1$ wieder eine Primzahl, dann ist nichts zu zeigen, da $n+1$ direkt ein Produkt von Primzahlen ist.

ii) Sei $n+1$ keine Primzahl, dann existieren mindestens zwei Zahlen p und q mit $2 \leq p, q < n+1$ und $p \cdot q = n+1$. Nach Induktionsvoraussetzung sind dann p und q wieder als Produkt von Primzahlen darstellbar. Etwa $p = p_1 \cdot p_2 \cdot \ldots \cdot p_s$ und $q = q_1 \cdot q_2 \cdot \ldots \cdot q_t$. Damit ist aber $n+1 = p \cdot q = p_1 \cdot p_2 \cdot \ldots \cdot p_s \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_t$ ein Produkt von Primzahlen. #

Induktionsbeweise treten z.B. bei der Analyse von Programmen immer wieder auf und spielen deshalb für die Informatik eine ganz besonders wichtige Rolle.

C.6.2. Induktive Definitionen

Das Induktionsprinzip kann man aber auch dazu verwenden, (Daten-)Strukturen formal zu spezifizieren. Dies macht diese Technik für Anwendungen in der Informatik besonders interessant. Dazu werden in einem ersten Schritt ($\triangleq$ Induktionsanfang) die „atomaren“ Objekte definiert und dann in einem zweiten Schritt die zusammengesetzten Objekte ($\triangleq$ Induktionsschritt). Diese Technik ist als *induktive Definition* bekannt.

Beispiel 111: *Die Menge der* binären Bäume *ist wie folgt definiert:*

(IA) *Ein einzelner Knoten w ist ein* Baum *und w ist die* Wurzel *dieses Baums.*
(IS) *Seien $T_1, T_2, \ldots, T_n$ Bäume mit den Wurzeln $k_1, \ldots, k_n$ und w ein einzelner neuer Knoten. Verbinden wir den Knoten w mit allen Wurzeln $k_1, \ldots, k_n$, dann entsteht ein neuer Baum mit der Wurzel w. Nichts sonst ist ein Baum.*

Beispiel 112: *Die Menge der* arithmetischen Ausdrücke *ist wie folgt definiert:*

(IA) *Jeder Buchstabe und jede Zahl ist ein arithmetischer Ausdruck.*
(IS) *Seien E und F Ausdrücke, so sind auch $E+F$, $E*F$ und $[E]$ Ausdrücke. Nichts sonst ist ein Ausdruck.*

D.h. x, $x+y$, $[2*x+z]$ sind arithmetische Ausdrücke, aber beispielsweise sind $x+$, yy, $][x+y$ sowie $x+*z$ keine arithmetischen Ausdrücke im Sinn dieser Definition.

Beispiel 113: *Die Menge der* aussagenlogischen Formeln *ist wie folgt definiert:*

(IA) *Jede aussagenlogische Variable $x_1, x_2, x_3, \ldots$ ist eine aussagenlogische Formel.*
(IS) *Seien H_1 und H_2 aussagenlogische Formeln, so sind auch $(H_1 \wedge H_2)$, $(H_1 \vee H_2)$, $\neg H_1$, $(H_1 \leftrightarrow H_2)$, $(H_1 \rightarrow H_2)$ und $(H_1 \oplus H_2)$ aussagenlogische Formeln. Nichts sonst ist eine aussagenlogische Formel.*

Bei diesen Beispielen ahnt man schon, dass solche Techniken zur präzisen und eleganten Definition von Programmiersprachen und Dateiformaten gute Dienste leisten. Es zeigt sich, dass die im Compilerbau verwendeten Chomsky-Grammatiken eine andere Art von induktiven Definitionen darstellen. Darüber hinaus bieten induktive Definitionen noch weitere Vorteile, denn man kann oft relativ leicht Induktionsbeweise konstruieren, die Aussagen über induktiv definierte Objekte belegen / beweisen.

C.6.3. Die strukturelle Induktion

Satz 114: *Die Anzahl der öffnenden Klammern eines arithmetischen Ausdrucks stimmt mit der Anzahl der schließenden Klammern überein.*

Es ist offensichtlich, dass diese Aussage richtig ist, denn in Ausdrücken wie $(x + y)/2$ oder $x + ((y/2) * z)$ muss ja zu jeder öffnenden Klammer eine schließende Klammer existieren. Der nächste Beweis verwendet diese Idee um die Aussage von Satz 114 mit Hilfe einer *strukturellen Induktion* zu zeigen.
Beweis: Wir bezeichnen die Anzahl der öffnenden Klammern eines Ausdrucks E mit $\#_[(E)$ und verwenden die analoge Notation $\#_](E)$ für die Anzahl der schließenden Klammern.
(IA) Die einfachsten Ausdrücke sind Buchstaben und Zahlen. Die Anzahl der öffnenden und schließenden Klammern ist in beiden Fällen gleich 0.
(IV) Sei E ein Ausdruck, dann gilt $\#_[(E) = \#_](E)$.
(IS) Für einen Ausdruck $E+F$ gilt $\#_[(E+F) = \#_[(E)+\#_[(F) \overset{\text{IV}}{=} \#_](E)+\#_](F) = \#_](E+F)$. Völlig analog zeigt man dies für $E * F$. Für den Ausdruck $[E]$ ergibt sich $\#_[([E]) = \#_[(E)+1 \overset{\text{IV}}{=} \#_](E) + 1 = \#_]([E])$. In jedem Fall ist die Anzahl der öffnenden Klammern gleich der Anzahl der schließenden Klammern. #

Mit Hilfe von Satz 114 können wir nun leicht ein Programm entwickeln, das einen Plausibilitätscheck (z.B. direkt in einem Editor) durchführt und die Klammern zählt, bevor die Syntax von arithmetischen Ausdrücken überprüft wird. Definiert man eine vollständige Programmiersprache induktiv, dann werden ganz ähnliche Induktionsbeweise möglich, d.h. man kann die Techniken aus diesem Beispiel relativ leicht auf die Praxis der Informatik übertragen.

Man überlegt sich leicht, dass die natürlichen Zahlen auch induktiv definiert werden können (vgl. Peano-Axiome). Damit zeigt sich, dass die vollständige Induktion eigentlich nur ein Spezialfall der strukturellen Induktion ist.

Index

Symbole

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

Literatur

[Bre95] H. Breuer. *dtv-Atlas zur Informatik*. Deutscher Taschenbuch Verlag, 1995.

[Can95] G. Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

[GK05] G. Saake and K. Sattler. *Algorithmen und Datenstrukturen*. dpunkt.verlag, 2005.

[GPV+99] G. Ausiello, P. Crescenzi, V. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability*. Springer Verlag, 1999.

[Gru99] J. Gruska. *Quantum Computing*. McGraw-Hill, 1999.

[Hom08] M. Homeister. *Quantum Computing verstehen*. Vieweg, 2008.

[Knu97] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 3 edition, 1997.

[MD79] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[Pău98] G. Păun. *Computing with Bio-Molecules*. Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer Verlag, Sinpapore, 1998.

[RP95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Sch01] U. Sch"oning. *Algorithmik*. Spektrum Verlag, 2001.

[Sed97] R. Sedgewick. *Algorithmen in C*. Addison-Wesley, 1997.

[TCRC01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.