

# Diskrete und Geometrische Algorithmen

Stefan Hetzl

[stefan.hetzl@tuwien.ac.at](mailto:stefan.hetzl@tuwien.ac.at)

TU Wien

WS 2020/21



# Inhaltsverzeichnis

<b>Vorwort</b>	<b>v</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Berechnungsprobleme und Algorithmen . . . . .	1
1.2 Korrektheitsbeweise . . . . .	2
1.3 Aufwandsabschätzung . . . . .	7
<b>2 Elementare Kombinatorik</b>	<b>13</b>
2.1 Abzählprobleme . . . . .	13
2.2 Graphen . . . . .	16
2.3 Bäume . . . . .	18
<b>3 Teile und Herrsche</b>	<b>23</b>
3.1 Sortieren durch Verschmelzen . . . . .	23
3.2 Matrixmultiplikation . . . . .	25
3.3 Dichtestes Punktepaar . . . . .	29
<b>4 Rekursionsgleichungen</b>	<b>35</b>
4.1 Lineare Rekursionsgleichungen erster Ordnung . . . . .	35
4.2 Lineare Rekursionsgleichungen $k$ -ter Ordnung . . . . .	36
4.3 Die Substitutionsmethode . . . . .	39
4.4 Die Rekursionsbaummethode . . . . .	41
4.5 Die Mastermethode . . . . .	42
<b>5 Datenstrukturen</b>	<b>49</b>
5.1 Das Wörterbuchproblem . . . . .	49
5.2 Suchbäume . . . . .	51
5.3 AVL-Bäume . . . . .	58
5.4 Stapel und Warteschlangen . . . . .	62
5.5 Halden und Prioritätswarteschlangen . . . . .	63
<b>6 Suchen und Sortieren in Graphen</b>	<b>69</b>
6.1 Breitensuche und Tiefensuche . . . . .	69
6.2 Topologisches Sortieren . . . . .	71
<b>7 Gierige Algorithmen</b>	<b>75</b>
7.1 Der Algorithmus von Kruskal . . . . .	75
7.2 Der Algorithmus von Prim . . . . .	78
7.3 Der Algorithmus von Dijkstra . . . . .	80
7.4 Matroide . . . . .	81
7.5 Das Knotenüberdeckungsproblem . . . . .	83

<b>8</b>	<b>Dynamische Programmierung</b>	<b>87</b>
8.1	Das Stabzerlegungsproblem . . . . .	89
8.2	Segmentierte Methode der kleinsten Quadrate . . . . .	90
<b>9</b>	<b>Randomisierung</b>	<b>93</b>
9.1	Randomisierung der Eingabe . . . . .	93
9.2	Quicksort . . . . .	95
9.3	Eine untere Schranke auf Sortieralgorithmen . . . . .	98
9.4	Primzahltests . . . . .	101
9.5	Der RSA-Algorithmus . . . . .	103
<b>10</b>	<b>Lineare Optimierung</b>	<b>105</b>
10.1	Einführung . . . . .	105
10.2	Reduktionen auf lineare Optimierung . . . . .	107
10.3	Der Simplex-Algorithmus . . . . .	111
	<b>Problemverzeichnis</b>	<b>121</b>
	<b>Algorithmenverzeichnis</b>	<b>123</b>
	<b>Literaturverzeichnis</b>	<b>125</b>

# Vorwort

Dieses Skriptum begleitet die an der TU Wien im WS 2020/21 gehaltene Vorlesung *Diskrete und geometrische Algorithmen*. Bis auf einzelne geringfügige Korrekturen ist es identisch mit der während des Semester Einheit für Einheit erstellten Version.

Bei dieser Vorlesung handelt es sich um eine Einführung in den Entwurf und die Analyse von Algorithmen für Studenten der Mathematik. Die Struktur der Vorlesung orientiert sich dabei weitgehend an den zentralen Designprinzipien für Algorithmen, wie etwa teile-und-herrsche, dynamische Programmierung, oder gierige Algorithmen. Zum Zweck einer ausgewogenen Darstellung des Gebiets spielen auch die Graphentheorie sowie fundamentale Datenstrukturen eine wichtige Rolle als Querschnittsthemen. Die mathematischen Grundlagen diskreter Algorithmen, wie etwa elementare Kombinatorik oder Rekursionsgleichungen, werden, soweit es für die Vorlesung sinnvoll ist, gründlich besprochen. Bei der Auswahl konkreter Algorithmen wurden, wo möglich, mathematische Fragestellungen gewählt wie z.B. Matrixmultiplikation, lineare Optimierung, geometrische Probleme, etc.

Als ergänzende Literatur können zum Thema der Vorlesung die englischsprachigen Lehrbücher [2, 5, 3] sowie die deutschsprachigen Lehrbücher [6, 7] empfohlen werden. Zur vertiefenden Lektüre kann zur Analyse von Algorithmen [8], zur Kryptographie [1], zur linearen Optimierung [9] sowie zu geometrischen Algorithmen [4] empfohlen werden.



# Kapitel 1

## Einführung

### 1.1 Berechnungsprobleme und Algorithmen

**Definition 1.1.** Ein *Berechnungsproblem* ist eine Relation der Form  $P \subseteq X \times Y$  so dass für alle  $x \in X$  ein  $y \in Y$  existiert mit  $(x, y) \in P$ .

Dabei stellen wir uns  $X$  als Menge der möglichen Eingaben vor,  $Y$  als Menge der möglichen Ausgaben und  $(x, y) \in P$  als die Aussage “bei Eingabe  $x$  ist  $y$  eine korrekte Ausgabe”. Oft werden wir statt Berechnungsproblem einfach nur Problem sagen. Ein Berechnungsproblem ist aber zu unterscheiden von einem mathematischen Problem, bei welchem es sich (üblicherweise) um eine Frage der Form “Ist die Aussage ... wahr?” handelt. Beispiele für Berechnungsprobleme sind:

#### Bestimmung des ggT

Eingabe: positive ganze Zahlen  $n_1$  und  $n_2$

Ausgabe: der größte gemeinsame Teiler von  $n_1$  und  $n_2$

#### Sortierproblem

Eingabe: eine endliche Folge ganzer Zahlen  $(a_1, \dots, a_n)$

Ausgabe: eine Permutation  $(a_{\pi(1)}, \dots, a_{\pi(n)})$  so dass  $a_{\pi(1)} \leq \dots \leq a_{\pi(n)}$

#### Linearisierung

Eingabe: eine endliche partiell geordnete Menge  $(A, \leq)$

Ausgabe: eine totale Ordnung  $a_1, \dots, a_n$  der Elemente von  $A$  so dass  
 $a_i \leq a_j \Rightarrow i \leq j$

Wie man am dritten der obigen Beispiele sehen kann, muss ein Berechnungsproblem nicht unbedingt eine eindeutige Lösung haben. Falls  $P \subseteq X \times Y$  ein Problem ist, dann heißt jedes  $x \in X$  *Instanz von  $P$* , beispielsweise ist  $(3, 7, 2, 5, 8)$  eine Instanz des Sortierproblems.

**Definition 1.2.** Ein *Algorithmus* ist eine wohldefinierte Rechenvorschrift.

Wir geben hier keine präzisere Definition des Begriffs Algorithmus. Dies zu tun würde im Wesentlichen auf die Definition einer Programmiersprache hinauslaufen und damit am Thema dieser Vorlesung vorbeigehen. Wir werden konkrete Algorithmen in *Pseudocode* angeben, d.h. in einer der Situation angepassten Mischung aus natürlicher Sprache und üblichen Anweisungen und Kontrollstrukturen einer Programmiersprache wie Schleifen, Verzweigungen, usw. Wir verlangen auch nicht formell dass jeder Algorithmus *terminiert*, d.h. dass er für jede Eingabe nach endlicher Zeit stoppt. Allerdings werden wir in dieser Vorlesung fast ausschließlich terminierende Algorithmen betrachten.

Wir kennen bereits viele Algorithmen, zum Beispiel Algorithmen zur Addition und Multiplikation zweier natürlicher Zahlen in Dezimaldarstellung, wie sie in der Volksschule gelehrt werden, den Algorithmus zur Division mit Rest von Polynomen, den euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme, usw. Algorithmen sind aus der Mathematik nicht wegzudenken. Anders als bisher werden wir in dieser Vorlesung ein systematisches Studium von Algorithmen betreiben. Dieses beschränkt sich nicht auf die bloße Definition und Verwendung von Algorithmen, sondern untersucht Fragen wie z.B. “Wie effizient ist dieser Algorithmus?” “Wie ist das Verhältnis zwischen zwei Algorithmen?” “Welche Ansätze zur Entwicklung guter Algorithmen gibt es?” “Ist dieser Algorithmus der beste zur Lösung dieses Problems? In welchem Sinn ist er das?” usw. Algorithmen werden also von einem Mittel zur Lösung von Problemen zu einem Objekt unserer Untersuchungen.

Oft operieren Algorithmen auf umfangreichen Daten. Ein wichtiger Aspekt ist dann die Frage in welcher Form bzw. Struktur die Daten gespeichert werden. Man spricht in diesem Zusammenhang auch von *Datenstrukturen*. In dieser Vorlesung werden wir eine Reihe nützlicher und effizienter Datenstrukturen kennenlernen. Eine der einfachsten und gleichzeitig wichtigsten Datenstrukturen ist das *Datenfeld* (engl. *array*). Mathematisch handelt es sich dabei um eine endliche Folge. Die Elemente der Folge werden in aufsteigender Reihenfolge im Speicher abgelegt. Wir schreiben  $A, B, \dots$  für Datenfelder,  $A[i]$  für das  $i$ -te Element, der kleinste Index eines Datenfelds ist 1, wir schreiben  $A.Länge$  für die Länge des Datenfelds (d.h. für die Anzahl von Elementen, die es enthält). Die Notation  $A.Länge$  wird in der Informatik verwendet, um das Attribut *Länge* des Objekts  $A$  zu notieren. Manche Attribute (wie dieses) können nur gelesen werden (engl. *read-only*), manche können auch geschrieben werden, d.h. in Zuweisungen verwendet werden. Was davon der Fall ist wird üblicherweise aus dem Kontext heraus klar sein.

Algorithmen werden verwendet um Berechnungsprobleme zu lösen. Damit meint man Folgendes:

**Definition 1.3.** Sei  $P \subseteq X \times Y$  ein Berechnungsproblem und  $\mathcal{A}$  ein Algorithmus. Wir sagen dass  $\mathcal{A}$  *das Problem*  $P$  löst falls für jede Instanz  $x \in X$  gilt dass  $y = \mathcal{A}(x)$  die Eigenschaft  $(x, y) \in P$  hat.

Oft wird aus dem Kontext heraus klar sein, welches Berechnungsproblem wir lösen wollen, dann sprechen wir einfach von der *Korrektheit* eines Algorithmus.

## 1.2 Korrektheitsbeweise

Ein *Korrektheitsbeweis* ist ein Beweis der zeigt, dass ein bestimmter Algorithmus ein bestimmtes Berechnungsproblem löst. Um diesen Begriff zu illustrieren wollen wir zunächst ein einfaches Beispiel betrachten. Der folgende Algorithmus erhält ein Datenfeld (dessen Einträge Zahlen sind) als Eingabe und liefert eine Zahl als Ausgabe.

---

**Algorithmus 1** Prozedur P

---

```
1: Prozedur P( $A$ )
2:    $m := 0$ 
3:   Für  $i = 1, \dots, A.Länge$ 
4:      $m := m + A[i]$ 
5:   Ende Für
6:   Antworte  $m/A.Länge$ 
7: Ende Prozedur
```

---

Mit etwas Programmiererfahrung lässt sich leicht erkennen, dass Algorithmus 1 das folgende Berechnungsproblem löst:

<b>Arithmetisches Mittel</b>
------------------------------

Eingabe: eine endliche Folge $a_1, \dots, a_n$
--

Ausgabe: arithmetischer Mittelwert von $a_1, \dots, a_n$
--

Wir wollen diese Aussage nun präzise formulieren und beweisen. Ein wesentlicher Aspekt von Korrektheitsaussagen und Korrektheitsbeweisen besteht darin, dass wir es mit zwei unterschiedlichen Sprachebenen zu tun haben: einerseits die Sprachebene des Programms oder Pseudocodes und die in ihm vorkommenden *Programmvariablen*, andererseits die übliche mathematische Sprache in der wir Aussagen *über* den Pseudocode formulieren. Dieser Algorithmus erhält als Eingabe ein Datenfeld  $A$ , er benutzt die lokale Variable  $m$  und den Schleifenzähler  $i$ , der im Körper der Schleife ebenfalls eine lokale Variable ist. Die Programmvariablen sind also  $A$ ,  $m$  und  $i$ . Zur Verdeutlichung dieses Unterschieds verwenden wir für den Rest dieses Abschnitts für Programmvariablen *Schreibmaschinenschrift*, d.h. für dieses Beispiel:  $A, m, i$ . Wir können also formulieren:

**Satz 1.1.** *Algorithmus 1 ist korrekt, d.h. er löst das Berechnungsproblem “Arithmetisches Mittel”, d.h.  $P(A)$  antwortet mit  $\frac{\sum_{j=1}^{A.Länge} A[j]}{A.Länge}$ .*

Man beachte dass in obiger Formel die Variable  $j$  *keine* Programmvariable ist sondern eine Variable der üblichen mathematischen Sprache. Um diese Aussage formal zu beweisen benutzen wir eine *Schleifeninvariante*. Eine Invariante für eine bestimmte Schleife ist eine logische Aussage  $I$  über jene Programmvariablen, die zu Beginn eines Durchlaufs der Schleife verfügbar sind, mit den folgenden Eigenschaften:

1.  $I$  ist zu Beginn des ersten Durchlaufs der Schleife wahr und
2.  $I$  wird von der Schleife erhalten, d.h. wenn sie zu Beginn des  $i$ -ten Durchlaufs wahr ist, dann ist sie auch zu Beginn des  $i + 1$ -ten Durchlaufs wahr.

Damit ist eine Schleifeninvariante auch nach dem letzten Durchlauf der Schleife wahr. Sinnvollerweise wählt man die Invariante so, dass sie am Ende der Schleife eine Form hat, die für den weiteren Korrektheitsbeweis nützlich ist. Deshalb enthält ein Korrektheitsbeweis im Kontext einer Schleifeninvariante noch den weiteren Teil der

3. Verwendung der Invariante im Korrektheitsbeweis.

Auf diese Weise werden in einem Programmablauf mit  $n$  Durchläufen einer bestimmte Schleife zusätzlich zum Beginn und zum Ende des Programms noch  $n + 1$  weitere Zeitpunkte definiert: 1. der Beginn des 1. Schleifendurchlaufs,  $\dots$ ,  $n$ . der Beginn des  $n$ -ten Schleifendurchlaufs,  $n + 1$ . der Beginn des  $n + 1$ -ten Schleifendurchlaufs der die Schleife beendet. Es ist oft nützlich den Wert einer Programmvariablen  $x$  zum  $i$ -ten dieser Zeitpunkte mit  $x_i$  zu bezeichnen. Für Algorithmus 1 erhalten wir so die Werte  $m_k$ ,  $A_k$  und  $i_k$  für  $k = 1, \dots, n + 1$  wobei  $n = A.Länge$ . Nun sieht man leicht dass  $A_1 = \dots = A_{n+1}$  so dass wir den Index von  $A$  einfach weglassen. Außerdem gilt  $i_k = k$  für  $k = 1, \dots, n + 1$ .

*Beweis von Satz 1.1.* Unsere Schleifeninvariante ist:

$$I(A, m, i): m = \sum_{j=1}^{i-1} A[j].$$

Zunächst weisen wir nach, dass diese zu Beginn des ersten Durchlaufs der Schleife wahr ist. Dann ist  $k = 1$  und damit erhalten wir:

$$1. m_1 = 0 = \sum_{j=1}^0 A[j].$$

Im nächsten Schritt weisen wir nach, dass die Invariante durch die Schleife erhalten wird. Dazu nehmen wir (ähnlich einer Induktionshypothese) an dass die Invariante zu Beginn des  $i$ -ten Durchlaufs wahr ist, d.h.  $m_k = \sum_{j=1}^{i_k-1} A[j]$ , und erhalten:

$$2. m_{k+1} = m_k + A[i_k] = \sum_{j=1}^{i_k} A[j] = \sum_{j=1}^{i_{k+1}-1} A[j].$$

Um den Beweis abzuschließen betrachten wir den Beginn des  $n + 1$ -ten Schleifendurchlaufs. Bei diesem wird nach Inkrementierung von  $i$  festgestellt, dass die Schleife zu beenden ist und damit wird die Prozedur mit Zeile 6 fortgesetzt. Als Antwort der Prozedur erhalten wir also

$$3. m_{n+1}/n = \frac{\sum_{j=1}^n A[j]}{n}$$

was die Behauptung zeigt. □

Wie man an obigem Beweis exemplarisch sehen kann handelt es sich bei der Verwendung von Schleifeninvarianten um eine strukturierte Vorgehensweise zur Führung von Induktionsbeweisen über imperative Programme. Bei dieser Vorgehensweise verwendet man für jede in einem Algorithmus vorkommende Schleife eine Invariante. Wie sich diese Invarianten zueinander verhalten hängt vom Verhältnis der Schleifen zueinander ab, so entsprechen etwa zwei verschachtelte Schleifen einem verschalteten Induktionsbeweis mittels Schleifeninvarianten. Es ist möglich, in einem präzisen logischen Sinn zu zeigen, dass Beweise mittels Schleifeninvarianten ausreichend sind um Eigenschaften von imperativen Programmen nachzuweisen, d.h. dass jede wahre Aussage über ein Programm durch einen Beweis mit Schleifeninvarianten gezeigt werden kann.

Eines der am häufigsten auftretenden Berechnungsprobleme ist das Sortieren eines Datenfelds. Folglich sind Sortieralgorithmen sehr gründlich untersucht worden. Als nächstes Beispiel wollen wir hier einen ersten Sortieralgorithmus betrachten: Einfügesortieren (engl. *insertion sort*). Die Grundidee besteht darin, ein Datenfeld zu unterteilen in einen bereits sortierten Bereich (links) und einen noch nicht sortierten Bereich (rechts). Der sortierte Bereich wird dann sukzessive vergrößert, indem das erste Element des unsortierten Bereichs in den sortierten Bereich (an der richtigen Stelle) eingefügt wird. Am Ende ist der unsortierte Bereich leer und das Datenfeld damit vollständig sortiert. Die Idee kann wie folgt als Pseudocode notiert werden:

---

**Algorithmus 2** Einfügesortieren

---

```
1: Prozedur EINFÜGESORTIEREN( $A$ )
2:   Für  $j := 2, \dots, A.Länge$ 
3:      $x := A[j]$ 
4:      $i := j - 1$ 
5:     Solange  $i \geq 1$  und  $A[i] > x$ 
6:        $A[i + 1] := A[i]$ 
7:        $i := i - 1$ 
8:     Ende Solange
9:      $A[i + 1] := x$ 
10:  Ende Für
11: Ende Prozedur
```

---

*Beispiel 1.1.* siehe `bsp.einfuegesortieren.pdf`.

Um die Korrektheit von Einfügesortieren zu beweisen besprechen wir zunächst noch kurz einige Begriffe und Notationen die für Korrektheitsbeweise von Algorithmen im Allgemeinen von Bedeutung sind: Eine *Vorbedingung* eines Algorithmus ist eine Bedingung von der wir annehmen, dass sie beim Start des Algorithmus erfüllt ist. Eine *Nachbedingung* ist eine Aussage von der wir zeigen wollen, dass sie nach Ausführung des Algorithmus erfüllt ist falls die Eingabe die Vorbedingung erfüllt hat. Eine Korrektheitsaussage hat also oft die Form “Falls die Vorbedingung ... erfüllt ist, dann ist nach Ausführung des Algorithmus ... die Nachbedingung ...” erfüllt.

Beweise mittels Schleifeninvarianten werden oft systematisch in die oben erwähnten Punkte 1 bis 3 unterteilt. Damit ist an fast jeder Stelle klar auf den Wert zu welchem Zeitpunkt man sich bezieht wenn man eine Programmvariable erwähnt: bei 1 auf den Beginn der Schleife und bei 3 auf das Ende der Schleife. Die einzige Ausnahme ist 2 wo man sich auf den Wert zu zwei unterschiedlichen Zeitpunkte bezieht: zu Beginn der  $i$ -ten Iteration und zu Beginn der  $i + 1$ -ten Iteration. Eine gebräuchliche Notation für diese Situation besteht darin den Wert einer Programmvariablen  $X$  zu Beginn der  $i$ -ten Iteration einfach als  $X$  zu notieren und jenen zu Beginn der  $i + 1$ -ten Iteration als  $X'$ . Auch Varianten davon wie zum Beispiel  $X_{\text{alt}}$  und  $X_{\text{neu}}$  sind in Gebrauch.

Wir wollen nun in dieser kompakteren Notation die Korrektheit von Einfügesortieren beweisen. Dazu beginnen wir mit der folgenden Aussage über die innere Schleife.

**Lemma 1.1.** *Sei  $\text{EINFÜGEN}(A, i, x)$  eine Abkürzung für die Zeilen 5-9 von Algorithmus 2. Dann gilt für  $\text{EINFÜGEN}(A, i, x)$  unter der Vorbedingung (V)*

$i_0 = i, n = A.Länge, A[1, \dots, i_0] = m_1, \dots, m_{i_0}$  ist sortiert  
und  $A[i_0 + 1, \dots, n] = m_{i_0+1}, \dots, m_n$

die Nachbedingung (N)

$A[1, \dots, i_0 + 1] = m_1, \dots, m_i, x, m_{i+1}, \dots, m_{i_0}$  ist sortiert  
und  $A[i_0 + 2, \dots, n] = m_{i_0+2}, \dots, m_n$ .

*Beweis.* Der Körper der Schleife in  $\text{EINFÜGEN}(A, i, x)$  induziert die Abbildung  $(A, i) \mapsto (A', i')$  wobei

$$A'[k] = \begin{cases} A[i] & \text{falls } k = i + 1 \\ A[k] & \text{sonst} \end{cases} \quad \text{und} \quad i' = i - 1.$$

Für diese Schleife verwenden wir die Invariante  $I(A, i)$ :

$i \leq i_0$ ,  
 $A[1, \dots, i_0 + 1] = m_1, \dots, m_i, m_{i+1}, m_{i+1}, \dots, m_{i_0}$ ,  
 $A[i_0 + 2, \dots, n] = m_{i_0+2}, \dots, m_n$   
 und falls  $i < i_0$  dann  $x < m_{i+1}$ .

und argumentieren wie folgt:

1. Zu Beginn der Schleife folgt  $I(A, i)$  unmittelbar aus (V).
2.  $I(A, i)$  impliziert  $I(A', i')$  da erstens  $i' = i - 1 \leq^{I(A, i)} i_0 - 1 < i_0$  und damit bleibt  $A[i_0 + 2, \dots, n]$  unverändert. Weiters ist

$$\begin{aligned}
 A'[1, \dots, i_0 + 1] &= A[1, \dots, i], A[i], A[i + 2, \dots, i_0 + 1] \\
 &=^{I(A, i)} m_1, \dots, m_i, m_i, m_{i+1}, \dots, m_{i_0} \\
 &= m_1, \dots, m_{i'+1}, m_{i'+1}, \dots, m_{i_0}.
 \end{aligned}$$

Schließlich ist  $i' = i - 1 < i_0$ , also ist zu zeigen dass  $x < m_{i'+1} = m_i =^{I(A, i)} A[i]$  was unmittelbar aus der Wahrheit der Schleifenbedingung folgt.

3. Bei Beendigung der Schleife ist  $A[1, \dots, i_0 + 1] = m_1, \dots, m_i, m_{i+1}, m_{i+1}, \dots, m_{i_0}$  mit  $x < m_{i+1}$  und ( $i = 0$  oder ( $i \geq 1$  und  $A[i] = m_i \leq x$ )). Nach Ausführung von Zeile 9 ist damit  $A[1, \dots, i_0 + 1] = m_1, \dots, m_i, x, m_i + 1, \dots, m_{i_0}$  sortiert und damit ist (N) gezeigt.

□

**Satz 1.2.** *Einfügesortieren ist korrekt.*

*Beweis.* Genauer gesagt wollen wir zeigen dass für EINFÜGESORTIEREN(A) unter der Vorbedingung (V\*)

$$n = A.Länge \text{ und } A[1, \dots, n] = p_1, \dots, p_n$$

die Nachbedingung (N\*)

$$A[1, \dots, n] \text{ ist sortierte Permutation von } p_1, \dots, p_n$$

gilt. Dazu verwenden wir für die äußere Schleife die Invariante  $I(A, j)$ :

$$\begin{aligned}
 A[1, \dots, j - 1] &\text{ ist sortierte Permutation von } p_1, \dots, p_{j-1} \text{ und} \\
 A[j, \dots, n] &= p_j, \dots, p_n.
 \end{aligned}$$

und argumentieren wie folgt:

1. Zu Beginn der Schleife ist  $j = 2$  und damit folgt  $I(A, j)$  unmittelbar aus (V\*).
2. Aus  $I(A, j)$  folgt (V) von Lemma 1.1 da  $i_0 = j - 1$  und  $A[1, \dots, j - 1] = p_1, \dots, p_{j-1}$  sortiert ist. Also ist wegen Lemma 1.1 nach Ausführung von Zeile 9, und damit zu Beginn des nächsten Durchlaufs, die Nachbedingung (N) erfüllt. Also ist  $A'[1, \dots, j] = p_1, \dots, p_i, x, p_{i+1}, \dots, p_{j-1}$  und sortiert und  $A'[j+1, \dots, n] = A[j+1, \dots, n] = p_{j+1}, \dots, p_n$  woraus  $I(A', j')$  folgt da  $j = j' - 1$ .
3. Bei Beendigung der Schleife ist  $j = n + 1$  und damit folgt aus  $I(A, j)$  dass  $A[1, \dots, n]$  eine sortierte Permutation von  $p_1, \dots, p_n$  ist.

□

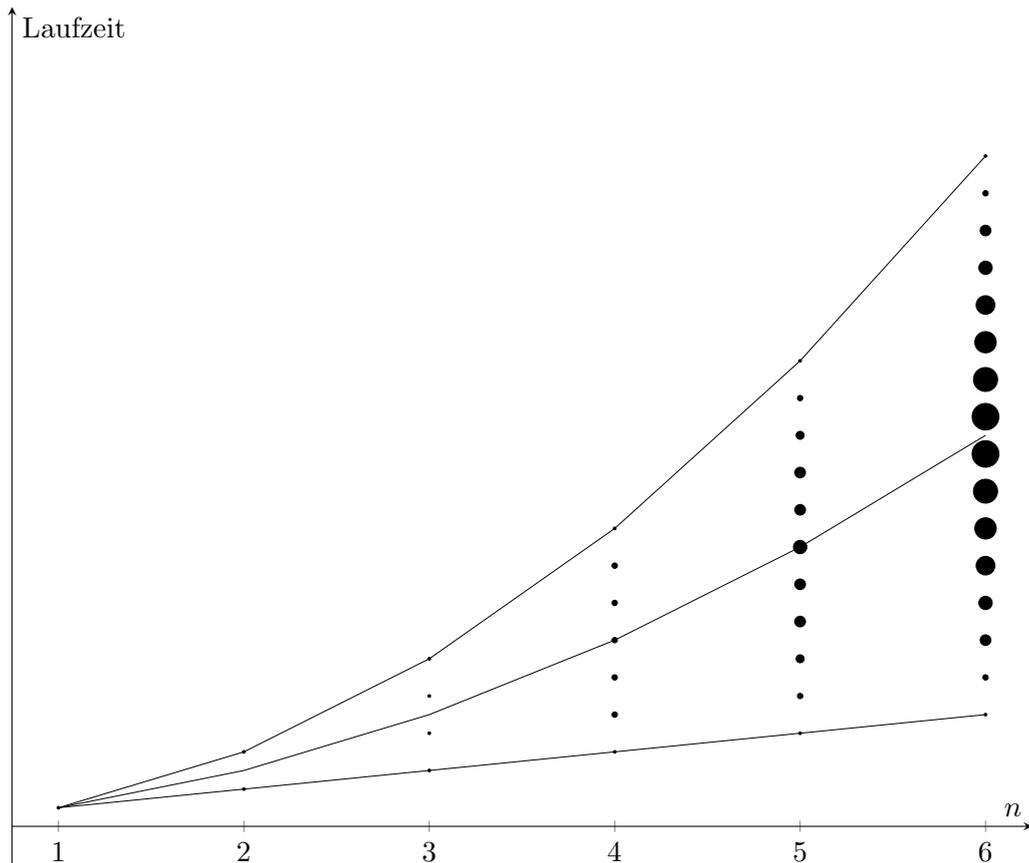


Abbildung 1.1: Laufzeit von Einfügesortieren für Datenfelder der Länge  $n \leq 6$

### 1.3 Aufwandsabschätzung

Der wichtigste Aspekt eines Algorithmus, neben seiner Korrektheit, ist typischerweise sein Ressourcenverbrauch, und hier vor allem: seine Laufzeit. Auch der Verbrauch anderer Ressourcen, zum Beispiel Speicherplatz, kann von Bedeutung sein, zentral ist aber in den allermeisten Situationen die Frage wie viel Zeit ein Algorithmus benötigt. Eine erste Antwort auf diese Frage bestünde darin, eine konkrete Implementierung des Algorithmus in einer bestimmten Programmiersprache anzufertigen, diese auf einem bestimmten Computer auf verschiedene Eingaben anzuwenden und die verbrauchte Zeit zu protokollieren. Derartige empirische Studien haben in der Informatik durchaus ihren Nutzen, allerdings haben sie die folgenden Nachteile:

1. Es ist unmöglich alle, typischerweise unendlich vielen, Eingaben auszuprobieren.
2. Die Ergebnisse hängen von der Implementierung, der Programmiersprache und vom Computer ab.

Um zu erklären wie man diese Nachteile durch geeignete Abstraktionen vermeiden kann betrachten wir zunächst ein Beispiel. In Abbildung 1.1 ist die Laufzeit von Einfügesortieren für Eingabedatenfelder der Länge  $n = 1, \dots, 6$  dargestellt. Da es bei Sortieralgorithmen nur auf die Reihenfolge der Elemente in der Ordnung ankommt, wurden hier für festes  $n$  als Eingabe nur die  $n!$  Permutationen von  $\{1, \dots, n\}$  betrachtet. Damit schränken wir unsere Betrachtung

auch auf Fälle ein wo keine Elemente mehrfach vorkommen. Die Größe eines Punktes ist proportional zur Anzahl der Fälle mit dieser Laufzeit. Wir sehen dass es für festes  $n$  (abhängig von der Permutation) verschiedene Laufzeiten gibt und dass die Laufzeiten mit steigendem  $n$  wachsen. Zusätzlich sind in Abbildung 1.1 drei Kurven eingezeichnet: die Laufzeit im besten Fall (engl. *best case*), die Laufzeit im schlechtesten Fall (engl. *worst case*), sowie die Laufzeit im Durchschnittsfall (engl. *average case*).

Ein Verhalten wie das in Abbildung 1.1 dargestellte ist typisch für die Praxis: üblicherweise wächst die Laufzeit mit der Größe der Eingabe. Um zu einer abstrakteren Darstellung der Laufzeit zu kommen betrachten wir sie also nicht als eine Funktion der Menge erlaubter Eingaben, sondern als eine Funktion der Eingabegröße. Um der Tatsache Rechnung zu tragen, dass es für eine fixe Eingabegröße unterschiedliche Laufzeiten gibt betrachtet man die drei Laufzeiten im besten, im schlechtesten, und im durchschnittlichen Fall.

Zur Illustration führen wir nun eine Analyse der Komplexität von Einfügesortieren durch. Um die Analyse von Algorithmen zu vereinfachen und von Details der Implementierung und der Hardware zu abstrahieren (sh. oben) trifft man üblicherweise die folgenden Annahmen:

1. Die Instruktionen werden sequentiell ausgeführt (was auf Mehrprozessor-Systemen nicht immer der Fall sein muss).
2. Der Zeitbedarf jeder "elementaren Operation" ist konstant<sup>1</sup>, also insbesondere ist er unabhängig von den Werten der Programmvariablen und wird nicht beeinflusst von Speichierachiekonzepten wie z.B. caching.
3. Wir arbeiten mit unbeschränkten Datentypen, also z.B. den natürlichen Zahlen und nicht, wie das in konkreten Implementierungen typischerweise der Fall ist mit, z.B.,  $\{0, \dots, 2^{64} - 1\}$ .
4. Wir nehmen an, dass wir mit den behandelten Zahlen exakt rechnen können. Der Einfluß von Rundungsfehlern bei der Verwendung von Gleitkommazahlen zur Darstellung reeller Zahlen spielt in der numerischen Mathematik eine wichtige Rolle, in dieser Vorlesung stellt er nur einen Nebenaspekt dar.

Wir können also voraussetzen dass für  $i = 2, \dots, 10$  eine Konstante  $c_i$  existiert, welche die Zeit angibt, die zur Ausführung von Zeile  $i$  benötigt wird. Sei  $A$  das Eingabedatenfeld und  $n$  die Länge von  $A$ . Dann belaufen sich die Kosten der Anwendung von Einfügesortieren auf das Datenfeld  $A$  auf

$$T(A) = (c_2 + c_{10})n + (c_3 + c_4 + c_9)(n - 1) + (c_5 + c_8) \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

wobei  $t_j$  angibt, wie oft die Bedingung der inneren Schleife überprüft wird, wenn der äußere Schleifenzähler  $j$  ist. Die  $t_j$  hängen offensichtlich von  $A$  ab. Was sind nun mögliche Werte für die  $t_j$  ?

Im besten Fall ist das Eingabedatenfeld  $A$  bereits sortiert. Dann gilt nämlich  $t_j = 1$  für  $j = 2, \dots, n$  und wir erhalten

$$T_b(n) = (c_2 + c_{10})n + (c_3 + c_4 + c_5 + c_8 + c_9)(n - 1),$$

---

<sup>1</sup>Wir geben hier keine präzise Definition davon, was als elementare Operation zu verstehen ist. Jedenfalls dazu gehören: arithmetische Operationen wie Addition und Multiplikation, schreibender und lesender Zugriff auf Variablen sowie die Ausführung von Kontrollstrukturen wie Bedingungen, Schleifenköpfen, etc.

d.h. also  $T_b(n) = kn + l$  für geeignete Konstanten  $k$  und  $l$ . Mit anderen Worten: die Laufzeit hängt linear von der Größe der Eingabe ab.

Im schlechtesten Fall muss jedes  $A[j]$  mit *allen* Elementen in  $A[1], \dots, A[j-1]$  verglichen werden. Das geschieht dann wenn das Eingabedatenfeld absteigend sortiert ist. Dann ist also  $t_j = j$  für  $j = 2, \dots, n$  und wir erhalten  $\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$  sowie  $\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$  aus der gaußschen Summenformel und damit

$$T_s(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + \frac{c_8}{2}\right)n^2 + (c_2 + c_3 + c_4 + c_9 + c_{10} + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + \frac{c_8}{2})n - c_3 - c_4 - c_5 - c_8 - c_9,$$

d.h. also  $T_s(n) = kn^2 + ln + m$  für geeignete Konstanten  $k$ ,  $l$  und  $m$ . Im schlechtesten Fall hängt die Laufzeit also quadratisch von der Größe der Eingabe ab.

Bei der Analyse des Durchschnittsfalls eröffnet sich eine zusätzliche Schwierigkeit: es ist a priori nicht klar, welche Wahrscheinlichkeitsverteilung den Eingabedaten zugrunde liegt. Diese kann auch je nach Anwendung recht unterschiedlich ausfallen. Der Einfachheit halber arbeitet man oft mit einer (in geeignetem Sinn) uniformen Wahrscheinlichkeitsverteilung. In diesem Fall, der Sortierung von Datenfeldern, bedeutet das, dass wir für ein Eingabedatenfeld  $A[1], \dots, A[n]$  und seine Sortierung  $A[\pi(1)], \dots, A[\pi(n)]$  annehmen dass jede Permutation  $\pi \in S_n$  gleich wahrscheinlich ist. D.h. jedes  $\pi \in S_n$  tritt mit Wahrscheinlichkeit  $\frac{1}{n!}$  auf. Man bezeichnet diese Annahme auch als *Permutationsmodell*. Dieses Modell für den allgemeinen Fall setzt auch die in Abbildung 1.1 betrachteten Eingabedaten für beliebige  $n \geq 1$  fort. Wir verwenden die folgende Eigenschaft einer zufällig gewählten  $\pi \in S_n$ : Sei  $1 \leq i \leq j \leq n$ , dann ist

$$W(\pi(j) \text{ ist das } i\text{-t-größte Element in } \{\pi(1), \dots, \pi(j)\}) = \frac{1}{j}.$$

Diese Aussage werden wir im nächsten Kapitel beweisen. Die mittlere Anzahl von Aufrufen des inneren Schleifenkopfs ist dann also

$$\bar{t}_j = \frac{1}{j}1 + \frac{1}{j}2 + \dots + \frac{1}{j}j = \frac{1}{j} \frac{j(j+1)}{2} = \frac{j+1}{2}.$$

Damit erhalten wir

$$\begin{aligned} \sum_{j=2}^n \bar{t}_j &= \frac{1}{2} \sum_{j=3}^{n+1} j = \frac{(n+1)(n+2)}{4} - \frac{3}{2} = \frac{n^2 + 3n}{4} - 1, \\ \sum_{j=2}^n (\bar{t}_j - 1) &= \frac{1}{2} \sum_{j=2}^n (j-1) = \frac{n(n-1)}{4} \text{ und} \\ T_d(n) &= \left(\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4} + \frac{c_8}{4}\right)n^2 \\ &\quad + (c_2 + c_3 + c_4 + c_9 + c_{10} + \frac{3c_5}{4} - \frac{c_6}{4} - \frac{c_7}{4} + \frac{3c_8}{4})n \\ &\quad - c_3 - c_4 - c_5 - c_8 - c_9, \end{aligned}$$

d.h. also  $T_d(n) = kn^2 + ln + m$  für geeignete Konstanten  $k$ ,  $l$  und  $m$ .

Mit dieser Analyse von Einfügesortieren haben wir also in einem gewissen Sinn eine Darstellung der Laufzeit auf *allen* Eingaben erhalten, womit wir das oben angesprochene erste Problem als behandelt betrachten wollen.

Was das zweite Problem angeht ist die entscheidende Beobachtung, dass sich die Wahl einer Programmiersprache, eines Compilers, etc. nur auf die  $c_i$  auswirkt, nicht aber darauf wie die

Laufzeit von  $n$  abhängt. Insbesondere haben wir rechnerisch gezeigt, dass diese Abhängigkeit, wie man aufgrund von Abbildung 1.1 bereits vermuten könnte, im besten Fall linear ist und im schlechtesten und durchschnittlichen Fall quadratisch. Um diese Information auf formale Weise darzustellen verwendet man die Landau-Symbole, auch “Groß-O-Notation” genannt ( $O(f)$  steht für “Ordnung von  $f$ ”).

**Definition 1.4.** Sei  $f : \mathbb{N} \rightarrow \mathbb{R}$  eine Funktion. Wir definieren

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|\}.$$

Falls  $g \in O(f)$  sagen wir dass  $f$  eine *asymptotisch obere Schranke* von  $g$  ist. Sei weiters

$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 : c \cdot |f(n)| \leq |g(n)|\}.$$

Falls  $g \in \Omega(f)$  sagen wir dass  $f$  eine *asymptotisch untere Schranke* von  $g$  ist. Schließlich definieren wir noch

$$\Theta(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, d > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 : c \cdot |f(n)| \leq |g(n)| \leq d \cdot |f(n)|\}.$$

Falls  $g \in \Theta(f)$  sagen wir dass  $f$  eine *asymptotisch scharfe Schranke* von  $g$  ist.

*Beispiel 1.2.* Seien  $c, d, e > 0$ , dann ist  $cn^2 + dn + e = \Theta(n^2)$ . Einerseits ist nämlich  $cn^2 + dn + e \leq (c + d + e)n^2$  für  $n \geq 1$ . Andererseits ist auch  $cn^2 \leq cn^2 + dn + e$  für  $n \geq 0$ .

In dieser Vorlesung wird diese Notation meist für nicht-negative Funktionen verwendet werden; dann sind die Betragsstriche überflüssig. In der Literatur wird diese Notation oft sinngemäß auch für  $f : \mathbb{R} \rightarrow \mathbb{R}$  oder Funktionen anderen Typs verwendet. Deshalb wird in der Literatur häufig der Typ von  $f$  gar nicht erst angegeben. Zu dieser Definition lassen sich unmittelbar die folgenden Beobachtungen machen.

**Lemma 1.2.** Für alle Funktionen  $f, g$  gilt:

1.  $\Theta(f) = O(f) \cap \Omega(f)$
2.  $g \in O(f)$  genau dann wenn  $f \in \Omega(g)$
3.  $g \in O(f)$  genau dann wenn  $\limsup_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| < \infty$
4.  $g \in \Omega(f)$  genau dann wenn  $\liminf_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| > 0$

wobei wir voraussetzen dass  $f$  und  $g$  nur endlich viele Nullstellen haben.

*Beweis.* 1. folgt direkt aus der Definition. Für 2. sei  $c > 0, n_0 \in \mathbb{N}$  so dass  $\forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|$ . Sei  $d = \frac{1}{c}$ , dann gilt  $\forall n \geq n_0 : d \cdot |g(n)| \leq |f(n)|$ , d.h. also  $f \in \Omega(g)$ . Für die Gegenrichtung setzen wir  $c = \frac{1}{d}$ .

Für 3. sei wieder  $c > 0, n_0 \in \mathbb{N}$  so dass  $\forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|$ , d.h.  $\left| \frac{g(n)}{f(n)} \right| \leq c$ . Also ist  $\left\{ \left| \frac{g(n)}{f(n)} \right| \mid n \geq n_0 \right\}$  im kompakten Intervall  $[0, c]$  enthalten, besitzt also einen größten Häufungspunkt in  $[0, c]$ . Für die Gegenrichtung sei  $n_0 - 1$  die größte Nullstelle von  $f$ . Dann ist  $\left( \left| \frac{g(n)}{f(n)} \right| \right)_{n \geq n_0}$  beschränkt: falls  $\left( \left| \frac{g(n)}{f(n)} \right| \right)_{n \geq n_0}$  nämlich unbeschränkt wäre, dann wäre  $\limsup_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| = \infty$ . Also  $\exists c > 0$  so dass  $\left| \frac{g(n)}{f(n)} \right| \leq c$ , d.h.  $|g(n)| \leq c \cdot |f(n)|$  für  $n \geq n_0$ . 4. folgt aus 2. und 3.  $\square$

Oft schreiben wir  $g(n) = O(f(n))$  statt  $g \in O(f)$  um das Rechnen mit Termen wie z.B.  $n^2 + O(n)$  zu ermöglichen.

**Satz 1.3.** Sei  $f(n) = \sum_{i=0}^k a_i n^i$ ,  $a_k \neq 0$ , dann  $f(n) = \Theta(n^k)$ .

*Beweis.* Man beachte dass

$$\left| \frac{\sum_{i=0}^k a_i n^i}{n^k} \right| = \left| a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \dots + \frac{a_0}{n^k} \right| \rightarrow |a_k| \text{ für } n \rightarrow \infty.$$

Daraus folgt mit Lemma 1.2 das Resultat. □

Wir sagen dass eine Funktion  $f$  von polynomialem Wachstum ist falls ein  $k \geq 1$  existiert so dass  $f(n) = O(n^k)$ .

Wir können nun die Laufzeit von Einfügesortieren im besten Fall angeben als  $\Theta(n)$ , jene im schlechtesten Fall als  $\Theta(n^2)$  und jene im (uniform verteilten) Durchschnittsfall ebenfalls als  $\Theta(n^2)$ . Oft werden wir uns bei Angabe der Laufzeit im schlechtesten Fall auf die obere Schranke, d.h.  $O(\cdot)$  beschränken und umgekehrt bei der Laufzeit im besten Fall auf die untere Schranke  $\Omega(\cdot)$ . Wenn wir also sagen dass der Algorithmus  $\mathcal{A}$  Laufzeit  $O(f)$  hat ist damit gemeint, dass er im schlechtesten Fall Laufzeit  $O(f)$  hat und analog für  $\Omega$  und den besten Fall.

Eng in Zusammenhang mit dieser asymptotischen Notation steht auch die Folgende:

**Definition 1.5.** Sei  $f : \mathbb{N} \rightarrow \mathbb{R}$  eine Funktion. Wir definieren

$$o(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|\}.$$

Falls  $g \in o(f)$  sagen wir dass  $g$  *asymptotisch kleiner als*  $f$  ist.

$$\omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c \cdot |f(n)| \leq |g(n)|\}.$$

Falls  $g \in \omega(f)$  sagen wir dass  $g$  *asymptotisch größer als*  $f$  ist.

**Lemma 1.3.** Für alle Funktionen  $f, g$  gilt:

1.  $o(f) \subseteq O(f)$
2.  $\omega(f) \subseteq \Omega(f)$
3.  $f \notin o(f)$
4.  $f \notin \omega(f)$
5.  $g \in o(f)$  genau dann wenn  $f \in \omega(g)$
6.  $o(f) \cap \omega(f) = \emptyset$
7.  $g \in o(f)$  genau dann wenn  $\lim_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| = 0$
8.  $g \in \omega(f)$  genau dann wenn  $\lim_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| = \infty$

wobei wir wieder annehmen, dass  $f$  und  $g$  nur endlich viele Nullstellen haben.

*Beweis.* 1. und 2. folgen unmittelbar aus der Definition. Für 3. setzen wir  $c = \frac{1}{2}$  und für 4. setzen wir  $c = 2$ . 5. kann analog zu Lemma 1.2 gelöst werden indem  $c$  auf  $\frac{1}{d}$  und  $d$  auf  $\frac{1}{c}$  gesetzt wird. 6. folgt durch Wahl geeigneter Konstanten ebenfalls direkt aus der Definition. Für 7. beachte man, dass die Definition von  $g \in o(f)$  geschrieben werden kann als  $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0: \left| \frac{g(n)}{f(n)} \right| \leq \varepsilon$ . Für 8. schreibt man die Definition von  $g \in \omega(f)$  als  $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0: \left| \frac{f(n)}{g(n)} \right| \leq \varepsilon$ , d.h.  $\left| \frac{g(n)}{f(n)} \right| \geq \frac{1}{\varepsilon}$ .  $\square$

*Beispiel 1.3.* Aus der Analysis wissen wir dass  $\lim_{n \rightarrow \infty} \frac{n^k}{c^n} = 0$  für alle  $k \in \mathbb{N}$  und  $c > 1$ , d.h. also  $n^k \in o(c^n)$ .

**Definition 1.6.** Für  $f, g : \mathbb{N} \rightarrow \mathbb{R}$  schreiben wir  $f \sim g$  genau dann wenn  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$ . Falls  $f \sim g$  sagen wir dass  $f$  und  $g$  *asymptotisch äquivalent* sind.

# Kapitel 2

## Elementare Kombinatorik

In diesem Kapitel werden wir einige elementare Begriffe und Resultate der Kombinatorik und der Graphentheorie kennen lernen bzw. wiederholen, da diese für das Studium von Algorithmen eine wichtige Grundlage bilden.

### 2.1 Abzählprobleme

Abzählprobleme sind klassische kombinatorische Fragestellungen. Bei einem Abzählproblem fragt man sich wie viele Elemente eine bestimmte endliche Menge hat. Abzählprobleme spielen bei der Analyse von Algorithmen oft eine wichtige Rolle. Dabei finden, für endliche Mengen  $A$  und  $B$ , oft die folgenden Überlegungen Anwendung:

1. Summenregel: Falls  $A \cap B = \emptyset$ , dann  $|A \cup B| = |A| + |B|$
2. Produktregel:  $|A \times B| = |A| \cdot |B|$
3. Gleichheitsregel: Falls eine Bijektion  $f : A \rightarrow B$  existiert, dann ist  $|A| = |B|$ .

Für eine endliche Menge  $A$  wird eine bijektive Abbildung  $\pi : A \rightarrow A$  auch als *Permutation* bezeichnet. Zur Erleichterung der Notation, geht man oft davon aus, dass  $A = \{1, \dots, n\}$ . Eine solche Permutation kann in einer *zweizeiligen Darstellung* als

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

geschrieben werden. Eine alternative Darstellung ist die *Zyklendarstellung* als

$$\pi = (a_1 \pi(a_1) \cdots \pi^{l_1-1}(a_1))(a_2 \pi(a_2) \cdots \pi^{l_2-1}(a_2)) \cdots (a_k \pi(a_k) \cdots \pi^{l_k-1}(a_k))$$

wobei  $l_i$  definiert ist als das kleinste  $l$  so dass  $\pi^l(a_i) = a_i$  und  $a_i$  so aus  $\{1, \dots, n\}$  gewählt wird, dass es in keinem der vorherigen Zyklen auftritt. Damit kommt also jedes  $a \in \{1, \dots, n\}$  genau ein Mal in der Zyklendarstellung vor.

*Beispiel 2.1.* Die Permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 5 & 1 & 3 & 4 \end{pmatrix}$$

wird in Zyklendarstellung als  $(164)(2)(35)$  geschrieben.

Die Anzahl von Permutationen von  $\{1, \dots, n\}$  ist  $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$ . Die Funktion  $n \mapsto n!$  kann auch rekursiv definiert werden durch  $0! = 1$  und  $(n+1)! = (n+1)n!$ . Die Menge aller Permutationen von  $\{1, \dots, n\}$  wird auch als  $S_n$  bezeichnet. Man kann sich leicht davon überzeugen, dass  $S_n$  mit der Komposition von Funktionen eine Gruppe bildet.

Eine *Variation ohne Wiederholung* besteht aus der  $k$ -fachen Auswahl eines von  $n$  Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen ohne Wiederholung ist  $n \cdot (n-1) \cdots (n-k+1) = \frac{n!}{(n-k)!}$ .

*Beispiel 2.2.* Für die ersten  $k$  Stellen einer Permutation  $\pi \in S_n$  werden  $k$  aus  $n$  Objekten ohne Wiederholung ausgewählt, es gibt also  $\frac{n!}{(n-k)!}$  Möglichkeiten. Für  $k = n$  ergibt sich dann wie gehabt  $\frac{n!}{(n-n)!} = n!$ .

Eine *Variation mit Wiederholung* besteht aus der  $k$ -fachen Auswahl eines von  $n$  Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen mit Wiederholung ist  $n \cdot n \cdots n$ , d.h.  $n^k$ .

*Beispiel 2.3.* Wenn ein Münzwurf entweder Kopf oder Zahl ergibt und eine Münze  $k$  mal hintereinander geworfen wird, gibt es insgesamt  $2^k$  verschiedene Versuchsausgänge. Die Menge der Versuchsausgänge, d.h., der  $k$ -fachen Wiederholung von "Kopf" oder "Zahl" steht in Bijektion zu der Menge der Zeichenketten der Länge  $k$  die nur aus 0 und 1 bestehen, notiert als  $\{0, 1\}^k$ . Diese wiederum steht in Bijektion zu den Teilmengen einer  $k$ -elementigen Menge. Somit gibt es auch davon jeweils genau  $2^k$ .

Eine *Kombination ohne Wiederholung* besteht aus der  $k$ -fachen Auswahl eines von  $n$  Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine  $k$ -elementige Teilmenge einer  $n$ -elementigen Menge. Die Anzahl der Kombinationen ohne Wiederholung ist  $\frac{n!}{(n-k)!k!} = \binom{n}{k}$ , sie ergibt sich durch die Anzahl  $\frac{n!}{(n-k)!}$  der Variationen ohne Wiederholung und der Überlegung, dass jeder Kombination  $k!$  Variationen entsprechen.

*Beispiel 2.4.* Bei der Multiplikation von  $(x+y) \cdots (x+y) = (x+y)^n$  müssen zur Bestimmung des Koeffizienten von  $x^k y^{n-k}$  alle Möglichkeiten in Betracht gezogen werden in  $n$  Faktoren  $k$  mal  $x$  zu wählen und die anderen  $n-k$  Mal  $y$ . Es gibt  $\binom{n}{k}$  solche Möglichkeiten, also ergibt sich der binomische Lehrsatz

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}.$$

Das erklärt auch warum  $\binom{n}{k}$  als *Binomialkoeffizient* bezeichnet wird.

*Beispiel 2.5.* Bei der Lottoziehung "6 aus 45" werden aus 45 Kugeln 6 Stück gezogen wobei die Reihenfolge für die Ermittlung des Gewinners egal ist. Es gibt also  $\binom{45}{6} = 8145060$  Möglichkeiten für den Ausgang der Ziehung.

Eine *Kombination mit Wiederholung* besteht aus der  $k$ -fachen Auswahl eines von  $n$  Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine  $k$ -elementige Multimenge<sup>1</sup> mit Träger  $\{1, \dots, n\}$ . Eine solche Multimenge kann geschrieben werden als Tupel  $(a_1, \dots, a_k)$  wobei  $1 \leq a_1 \leq \dots \leq a_k \leq n$ . Nun gibt es eine Bijektion von der Menge der  $k$ -elementigen Multimengen mit Träger  $\{1, \dots, n\}$  auf die  $k$ -elementigen Teilmengen von  $\{1, \dots, n+k-1\}$ , die durch

$$(a_1, \dots, a_k) \mapsto (a_1, a_2 + 1, \dots, a_k + k - 1)$$

<sup>1</sup>Sei  $X$  eine Menge. Eine Multimenge  $M$  mit Träger  $X$  ist gegeben durch ihre charakteristische Funktion  $\chi_M : X \rightarrow \mathbb{N}$ . Eine Multimenge kann also, anders als eine Menge, ein Element mehrfach enthalten. Man definiert  $|M| = \sum_{x \in X} \chi_M(x)$ .

gegeben ist. Dann ist nämlich  $1 \leq a_1 < a_2 + 1 < \dots < a_k + k - 1 \leq n + k - 1$ . Die Anzahl  $k$ -elementiger Teilmengen von  $\{1, \dots, n + k - 1\}$  kennen wir bereits:  $\binom{n+k-1}{k}$ .

*Beispiel 2.6.* Bei einem Brettspiel würfelt ein Spieler mit zwei Würfeln gleichzeitig. Bei den möglichen Würfeln handelt es sich um eine Kombination mit Wiederholung mit  $n = 6$  und  $k = 2$ . Es gibt also  $\binom{7}{2} = 21$  verschiedene Würfe.

**Lemma 2.1.** Sei  $1 \leq i \leq j \leq n$ , sei  $\pi \in S_n$  uniform gewählt, dann ist

$$W(\pi(j) \text{ ist das } i\text{-t-größte Element in } \{\pi(1), \dots, \pi(j)\}) = \frac{1}{j}.$$

*Beweis.* 1. Die Anzahl der Tupel  $(\pi(1), \dots, \pi(j))$  ist  $\frac{n!}{(n-j)!}$ . 2. Die Anzahl der Tupel  $(\pi(1), \dots, \pi(j))$  in denen  $\pi(j)$  am  $i$ -t-größten ist ergibt sich a) durch Auswahl der Menge  $\{\pi(1), \dots, \pi(j)\} \subseteq \{1, \dots, n\}$ , dafür gibt es  $\binom{n}{j} = \frac{n!}{(n-j)!j!}$  Möglichkeiten, b) durch Fixierung von  $\pi(j)$  als das  $i$ -t-größte Element und c) durch Auswahl einer Reihenfolge für  $\{\pi(1), \dots, \pi(j-1)\}$ , dafür gibt es  $(j-1)!$  Möglichkeiten, d.h. also insgesamt  $\frac{n!(j-1)!}{(n-j)!j!} = \frac{n!}{(n-j)!j}$ . Wir erhalten also

$$\frac{\text{günstige}}{\text{mögliche}} = \frac{n! \cdot (n-j)!}{(n-j)! \cdot j \cdot n!} = \frac{1}{j}.$$

□

Für zwei endliche Mengen  $A$  und  $B$  mit  $A \cap B = \emptyset$  gilt wie erwähnt  $|A \cup B| = |A| + |B|$ . Falls  $A \cap B \neq \emptyset$  werden durch  $|A| + |B|$  die Elemente im Durchschnitt doppelt gezählt. Durch Korrektur dieser Doppelzählung erhalten wir  $|A \cup B| = |A| + |B| - |A \cap B|$ . Im Fall von drei endlichen Mengen  $A$ ,  $B$  und  $C$  werden durch  $|A| + |B| + |C|$  alle Elemente die in zwei Mengen liegen zu oft gezählt. Eine erste Korrektur ergibt  $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C|$ . Nun werden aber die Elemente im Schnitt von drei Mengen gar nicht gezählt. Durch einen weiteren Korrekturschritt erhalten wir also  $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$ . Im Allgemeinen gilt:

**Satz 2.1** (Prinzip von Inklusion und Exklusion). Seien  $A_1, \dots, A_n$  endliche Mengen, dann ist

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq I \subseteq \{1, \dots, n\}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|$$

*Beweis.* Sei  $x \in \bigcup_{i=1}^n A_i$ ,  $S = \{i \in \{1, \dots, n\} \mid x \in A_i\}$  und  $s = |S|$ . Auf der rechten Seite wird  $x$  jetzt genau dann in einem Durchschnitt gezählt wenn  $I \subseteq S$  und zwar, je nach Kardinalität von  $I$  entweder positiv oder negativ. Das Element  $x$  wird also

$$\binom{s}{1} - \binom{s}{2} + \binom{s}{3} - \dots + (-1)^{s+1} \binom{s}{s} = \sum_{i=1}^s \binom{s}{i} (-1)^{i+1}$$

mal gezählt. Nun ist aber nach dem binomischen Lehrsatz  $\sum_{i=0}^s \binom{s}{i} (-1)^i = (1-1)^s = 0$ , sowie, nach Multiplikation mit  $-1$ , auch  $\sum_{i=0}^s \binom{s}{i} (-1)^{i+1} = 0$ . Weiters ist  $\sum_{i=0}^s \binom{s}{i} (-1)^{i+1} = -1 + \sum_{i=1}^s \binom{s}{i} (-1)^{i+1}$ . Damit wird  $x$  also  $\sum_{i=1}^s \binom{s}{i} (-1)^{i+1} = 1$  mal gezählt. □

Das *Schubfachprinzip* (engl. *pigeonhole principle*) besagt Folgendes: Seien  $A$  und  $B$  endliche Mengen mit  $|A| > |B|$ , dann existiert keine injektive Funktion  $f : A \rightarrow B$ . Es kann mit einem einfachen Induktionsargument bewiesen werden. Ein Beispiel für seine Anwendung liefert der folgende Satz.

**Satz 2.2.** Für jedes ungerade  $q \in \mathbb{N}$  existiert ein  $i \geq 1$  so dass  $q \mid 2^i - 1$ .

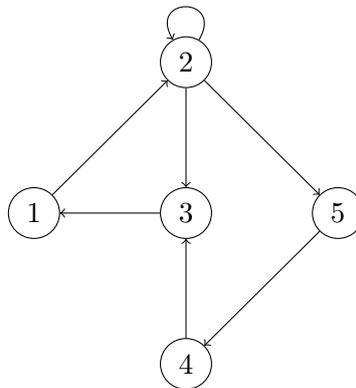
*Beweis.* Wir kürzen  $2^i - 1$  als  $a_i$  ab. Wir betrachten  $a_1, \dots, a_q \pmod{q}$ . Falls es darunter ein  $a_i$  gibt mit  $a_i \equiv 0 \pmod{q}$  dann sind wir fertig. Falls nicht, dann gibt es nach dem Schubfachprinzip  $i < j$  mit  $a_i \equiv a_j \pmod{q}$ , d.h.  $q \mid a_j - a_i$ . Nun ist aber  $a_j - a_i = 2^j - 1 - 2^i + 1 = 2^j - 2^i = 2^i(2^{j-i} - 1)$  und da  $q$  ungerade muss  $q \mid a_{j-i}$ , Widerspruch.  $\square$

## 2.2 Graphen

**Definition 2.1.** Ein *gerichteter Graph* ist ein Paar  $G = (V, E)$  wobei  $V$  eine beliebige Menge ist und  $E \subseteq V \times V$ .

Die Elemente von  $V$  heißen *Knoten*, die Elemente von  $E$  *Kanten*. Graphen werden oft aufgezichnet indem die Kanten als Pfeile zwischen den Knoten dargestellt werden.

*Beispiel 2.7.* Der gerichtete Graph  $G = (V, E)$  mit  $V = \{1, 2, 3, 4, 5\}$  und  $E = \{(1, 2), (2, 2), (2, 3), (2, 5), (3, 1), (4, 3), (5, 4)\}$  kann z.B. folgendermaßen gezeichnet werden.



Zwei Knoten  $x, y \in V$  heißen *adjazent* falls  $(x, y) \in E$  oder  $(y, x) \in E$ . Der *Ausgangsgrad* von  $v \in V$  ist  $d^+(v) = |\{w \in V \mid (v, w) \in E\}|$ . Der *Eingangsgrad* von  $v \in V$  ist  $d^-(v) = |\{u \in V \mid (u, v) \in E\}|$ . Ein Pfad ist eine endliche Folge  $v_1, \dots, v_n \in V$  mit  $(v_i, v_{i+1}) \in E$  für  $i = 1, \dots, n-1$  und  $i \neq j$  impliziert  $v_i \neq v_j$ .

Oft werden wir auch ungerichtete Graphen betrachten.

**Definition 2.2.** Ein *ungerichteter Graph* ist ein Paar  $(V, E)$  wobei  $V$  eine beliebige Menge ist und  $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$ .

Damit ist  $|E|$  in einem ungerichteten Graphen die Anzahl ungerichteter Kanten. Weiters definieren wir für  $v \in V$  den Grad von  $v$  als  $d(v) = |\{w \in V \mid \{v, w\} \in E\}|$ , es gibt also keinen getrennten Eingangs- und Ausgangsgrad mehr. Ein ungerichteter Graph kann als gerichteter Graph aufgefasst werden indem eine ungerichtete Kante  $\{x, y\}$  durch die gerichteten Kanten  $(x, y)$  und  $(y, x)$  ersetzt wird. In diesem Sinn stellen die ungerichteten Graphen den allgemeineren Begriff dar. Von nun an werden wir mit "Graph" immer einen ungerichteten Graphen meinen. Wenn wir gerichtete Graphen betrachten wollen, werden wir das explizit erwähnen.

Ein *Pfad* in einem Graphen  $G = (V, E)$  ist eine endliche Folge  $v_1, \dots, v_k \in V$  mit  $\{v_i, v_{i+1}\} \in E$  für  $i = 1, \dots, k-1$  und  $i \neq j$  impliziert  $v_i \neq v_j$ .  $G$  heißt *zusammenhängend* wenn es für alle  $v, w \in V$  einen Pfad von  $v$  nach  $w$  gibt.  $G$  heißt *vollständig* falls  $E = \{\{x, y\} \mid x, y \in V, x \neq y\}$ .

Graphen treten in einer Unzahl von Anwendungskontexten und Berechnungsproblemen auf, wobei man es in der Informatik naturgemäß üblicherweise mit endlichen Graphen zu tun hat. Beispiele für Situationen die durch Graphen modelliert werden können sind: Verkehrsnetze wobei die Knoten z.B. Städten entsprechen und die Kanten Zugverbindungen, das WWW wobei die Knoten Webseiten und die Kanten Hyperlinks entsprechen oder auch Landkarten wobei jedem Land ein Knoten entspricht und zwei Konten durch eine ungerichtete Kante verbunden sind falls sie aneinander grenzen. Beispiele für Berechnungsprobleme aus der Graphentheorie sind:

### Kürzester Pfad

Eingabe: endlicher zusammenhängender Graph  $G = (V, E)$ , Kostenfunktion  $c : E \rightarrow \mathbb{R}_{>0}$ ,  $s, t \in V$

Ausgabe: Pfad  $v_1, \dots, v_n$  mit  $v_1 = s$ ,  $v_n = t$  so dass  $\sum_{i=1}^{n-1} c(\{v_i, v_{i+1}\})$  minimal ist

### Problem des Handelsreisenden

(engl. *Travelling Salesman Problem (TSP)*)

Eingabe: endlicher vollständiger Graph  $G = (V, E)$ , Kostenfunktion  $c : E \rightarrow \mathbb{R}_{>0}$

Ausgabe: Pfad  $v_1, \dots, v_n, v_{n+1} = v_1$  mit  $\{v_1, \dots, v_n\} = V$  so dass  $\sum_{i=1}^n c(\{v_i, v_{i+1}\})$  minimal ist

### Knotenfärbung

Eingabe: endlicher Graph  $G = (V, E)$

Ausgabe: Abbildung  $f : V \rightarrow \{1, \dots, k\}$  so dass  $(x, y) \in E \Rightarrow f(x) \neq f(y)$  und  $k$  minimal ist

Wir werden später effiziente Algorithmen zur Bestimmung eines kürzesten Pfades sehen, insb. wird deren Laufzeit polynomial in der Größe der Eingabe sein. Für das Problem des Handelsreisenden oder jenes der Knotenfärbung sind keine Algorithmen mit polynomialer Laufzeit bekannt. Aus der Existenz eines solchen Algorithmus würde  $\mathbf{P} = \mathbf{NP}$  folgen und damit die Lösung eines der bedeutendsten offenen Probleme der Mathematik. Das  $\mathbf{P}$  vs.  $\mathbf{NP}$  Problem wird später noch etwas detaillierter besprochen werden.

Zur algorithmischen Behandlung von Graphen müssen diese (etwa im Speicher eines Computers) repräsentiert werden. Es gibt verschiedene Datenstrukturen zur Repräsentation eines Graphen.

**Definition 2.3.** Die *Adjazenzmatrix* eines gerichteten Graphen  $G = (\{1, \dots, n\}, E)$  ist die Matrix  $M = (m_{i,j})_{1 \leq i, j \leq n}$  wobei

$$m_{i,j} = \begin{cases} 1 & \text{falls } (i, j) \in E \\ 0 & \text{falls } (i, j) \notin E \end{cases}$$

*Beispiel 2.8.* Die Adjazenzmatrix des in Beispiel 2.7 angegebenen Graphen ist

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Eine Adjazenzmatrix wird im Speicher als ein Datenfeld von Datenfeldern abgelegt. Der für eine Adjazenzmatrix benötigte Speicherplatz ist  $\Theta(|V|^2)$ . Auf Basis einer Adjazenzmatrix kann von gegebenen Knoten  $v$  und  $w$  in Zeit  $\Theta(1)$  festgestellt werden ob der Graph die Kante  $(v, w)$  enthält. Das Durchlaufen aller von einem gegebenen Knoten  $v$  aus wegführenden Kanten benötigt  $\Theta(|V|)$  Zeit.

**Definition 2.4.** Die *Adjazenzliste* eines gerichteten Graphen  $G = (\{1, \dots, n\}, E)$  ist ein Datenfeld  $L$  der Länge  $n$  wobei  $L[i]$  die Liste jener  $j \in \{1, \dots, n\}$  ist für die  $(i, j) \in E$  gilt.

*Beispiel 2.9.* Die Adjazenzliste des in Beispiel 2.7 angegebenen Graphen ist:

- 1: 2
- 2: 2,3,5
- 3: 1
- 4: 3
- 5: 4

Der für eine Adjazenzliste benötigte Speicherplatz ist  $\Theta(|E|)$  was im schlechtesten Fall auch  $\Theta(|V|^2)$  ist. Für dünn besetzte Graphen, d.h. also solche die wesentlich weniger als  $|V|^2$  viele Kanten haben, stellt die Verwendung einer Adjazenzliste aber eine Speichersparnis dar. Auf Basis einer Adjazenzliste kann, gegeben Knoten  $v$  und  $w$ , in Zeit  $O(d^+(v))$  festgestellt werden, ob der Graph die Kante  $(v, w)$  enthält. Das Durchlaufen aller von einem gegebenen Knoten  $v$  wegführenden Kanten benötigt  $\Theta(d^+(v))$  Zeit.

*Beispiel 2.10.* Sei  $G = (V, E)$  ein Graph wobei  $V$  die Menge der Kreuzungen in einer Großstadt ist und  $(v, w) \in E$  falls eine Straße von  $v$  nach  $w$  führt ohne eine andere Kreuzung zu passieren. Der Graph  $G$  ist dünn besetzt, die Verwendung einer Adjazenzliste ist also empfehlenswert.

## 2.3 Bäume

Eine, insbesondere für Algorithmen, besonders wichtige Klasse von Graphen sind Bäume. Um Bäume näher zu untersuchen benötigen wir noch einige Begriffe: Sei  $G = (V, E)$  ein Graph. Ein Graph  $G' = (V', E')$  heißt *Teilgraph* von  $G$  falls  $V' \subseteq V$  und  $E' \subseteq E$ . Sei  $V' \subseteq V$ , dann ist  $G' = (V', E')$  der *von  $V'$  in  $G$  induzierte Graph* wobei  $E' = \{(v, w) \in E \mid v, w \in V'\}$ .

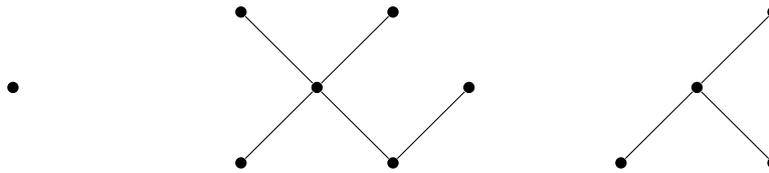
**Definition 2.5.** Sei  $G = (V, E)$  ein Graph.  $G' = (V', E')$  heißt *Zusammenhangskomponente* von  $G$  falls

1.  $V' \subseteq V$ ,
2.  $G'$  wird durch  $V'$  in  $G$  induziert,
3.  $G'$  ist zusammenhängend und
4. Für alle  $V''$  mit  $V' \subset V'' \subseteq V$  gilt: der durch  $V''$  in  $G$  induzierte Graph ist nicht zusammenhängend.

Aus dieser Definition folgt unmittelbar dass jeder Graph als disjunkte Vereinigung seiner Zusammenhangskomponenten dargestellt werden kann. Ein *Zyklus* ist ein Pfad  $v_1, \dots, v_k$  mit  $k \geq 3$  und  $\{v_k, v_1\} \in E$ .

**Definition 2.6.** Ein Graph  $G = (V, E)$  heißt *Baum* falls er zusammenhängend und zyklensfrei ist. Ein Graph  $G = (V, E)$  heißt *Wald* falls er zyklensfrei ist.

Beispiel 2.11. Ein Wald der aus 3 Bäumen besteht:

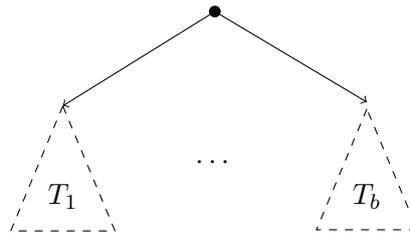


In dieser Definition eines Baums ist keine Wurzel ausgezeichnet, jeder Knoten kann als Wurzel designiert werden (je nachdem ändert sich dann die Form des Baums). Ein Wurzelbaum ist ein Tripel  $(V, E, r)$  wobei  $(V, E)$  ein Baum ist und  $r \in V$ . Ein Wurzelbaum kann als gerichteter Graph aufgefasst werden, indem, ausgehend von  $r$ , alle Kanten von  $r$  weg orientiert werden. Wurzelbäume werden uns, wie im folgenden Beispiel, oft auch in Form einer rekursiven Definition begegnen.

Beispiel 2.12. Sei  $b \geq 2$ . Ein *vollständiger Baum mit Arität  $b$*  und Tiefe 0 ist ein einzelner Knoten



Ein solcher Knoten heißt *Blatt*. Ein vollständiger Baum mit Arität  $b$  und Tiefe  $d + 1$  ist ein gerichteter Graph der Form



wobei  $T_1, \dots, T_b$  vollständige Bäume mit Arität  $b$  und Tiefe  $d$  sind. Jeder in der Wurzel beginnende Pfad in einem vollständigen Baum kann identifiziert werden mit einem Tupel  $(p_1, \dots, p_k) \in \{1, \dots, b\}^k$  wobei  $k \leq d$ . Also hat ein vollständiger Baum  $b^d$  *Blätter* sowie

$$b^0 + b^1 + \dots + b^d = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1}$$

Knoten. Ein vollständiger Binärbaum der Tiefe 3 hat also  $2^3 = 8$  Blätter und  $2^4 - 1 = 15$  Knoten.

Zum Zweck einer Charakterisierung der Bäume unter den Graphen machen wir zunächst noch einige vorbereitende Beobachtungen.

**Lemma 2.2.** Sei  $G = (V, E)$  ein endlicher, nicht-leerer, zyklensfreier Graph. Dann ist  $|E| \leq |V| - 1$ .

*Beweis.* Wir gehen mit Induktion nach  $|V|$  vor. Falls  $|V| = 1$ , dann muss  $|E| = 0$  sein. Sei nun  $|V| \geq 2$ . Falls  $E = \emptyset$  sind wir fertig. Falls  $E \neq \emptyset$ , dann existiert ein  $v \in V$  mit  $d(v) \geq 1$ . Seien  $\{v, w_1\}, \dots, \{v, w_k\}$  alle zu  $v$  inzidenten Kanten, dann gilt für alle  $i, j \in \{1, \dots, k\}$  mit  $i \neq j$ : jeder Pfad von  $w_i$  nach  $w_j$  enthält  $v$ , sonst würde nämlich  $G$  einen Zyklus enthalten. Für  $i = 1, \dots, k$  sei nun  $G_i = (V_i, E_i)$  die Zusammenhangskomponente von  $w_i$  im Graphen der

aus  $G$  entsteht wenn wir  $v$  sowie die Kanten  $\{v, w_1\}, \dots, \{v, w_k\}$  entfernen. Dann sind alle  $G_i$  zyklensfrei und mit der Induktionshypothese ist also

$$|E| = k + \sum_{i=1}^k |E_i| \stackrel{\text{IH}}{\leq} k + \sum_{i=1}^k (|V_i| - 1) = \sum_{i=1}^k |V_i| = |V| - 1.$$

□

**Lemma 2.3.** Sei  $G = (V, E)$  ein endlicher, nicht-leerer, zusammenhängender Graph. Dann ist  $|E| \geq |V| - 1$ .

*Beweis.* Wir gehen mit Induktion nach  $|V|$  vor. Falls  $|V| = 1$  ist das Resultat trivial. Sei nun  $|V| \geq 2$ . Da  $G$  zusammenhängend ist existiert ein Knoten  $v$  mit  $d(v) \geq 1$ . Seien  $\{v, w_1\}, \dots, \{v, w_k\}$  alle zu  $v$  inzidenten Kanten. Sei  $G' = (V', E')$  der Graph der aus  $G$  entsteht wenn wir  $v$  und sowie  $\{v, w_1\}, \dots, \{v, w_k\}$  löschen. Dann zerfällt  $G'$  in  $l \leq k$  Zusammenhangskomponenten  $(V_1, E_1), \dots, (V_l, E_l)$  und mit der Induktionshypothese gilt

$$|E| = k + \sum_{i=1}^l |E_i| \geq k + \sum_{i=1}^l (|V_i| - 1) \geq \sum_{i=1}^l |V_i| = |V| - 1.$$

□

**Definition 2.7.** Ein Graph  $G = (V, E)$  heißt *minimal zusammenhängend* falls er zusammenhängend ist und für alle  $E' \subset E$  gilt:  $(V, E')$  ist nicht zusammenhängend.

**Definition 2.8.** Ein Graph  $G = (V, E)$  heißt *maximal zyklensfrei* falls er zyklensfrei ist und für alle  $E' \supset E$  gilt:  $(V, E')$  enthält einen Zyklus.

**Satz 2.3.** Sei  $G = (V, E)$  ein endlicher, nicht-leerer Graph. Dann sind äquivalent:

1.  $G$  ist ein Baum.
2. Je zwei Knoten von  $G$  sind durch genau einen Pfad verbunden.
3.  $G$  ist minimal zusammenhängend.
4.  $G$  ist zusammenhängend und  $|E| = |V| - 1$ .
5.  $G$  ist zyklensfrei und  $|E| = |V| - 1$ .
6.  $G$  ist maximal zyklensfrei.

*Beweis.* 1.  $\Rightarrow$  2: Da  $G$  zusammenhängend ist, gibt es von  $u$  nach  $v$  einen Pfad. Angenommen es gibt zwei unterschiedliche Pfade von  $u$  nach  $v$ . Dann gibt es einen ersten Knoten  $w_1$  an dem sie sich unterscheiden und einen letzten Knoten  $w_2$  an dem sie sich unterscheiden. Die beiden Pfade von  $w_1$  nach  $w_2$  bilden dann einen Zyklus.

2.  $\Rightarrow$  3.: Da je zwei Knoten durch einen Pfad verbunden sind ist  $G$  zusammenhängend. Sei nun  $(u, v) \in E$ , dann ist  $u, v$  der einzige Pfad von  $u$  nach  $v$  und damit ist  $(V, E \setminus \{(u, v)\})$  nicht zusammenhängend.

3.  $\Rightarrow$  4.:  $G$  ist zusammenhängend und damit ist  $|E| \geq |V| - 1$  wegen Lemma 2.3. Es bleibt zu zeigen dass  $|E| \leq |V| - 1$  ist. Angenommen  $|E| > |V| - 1$  dann würde  $G$  wegen Lemma 2.2 einen Zyklus enthalten. Aus diesem könnte eine beliebige Kante gelöscht werden ohne den Zusammenhang zu zerstören,  $G$  wäre also nicht minimal zusammenhängend.

4.  $\Rightarrow$  5.: Angenommen  $G$  enthält einen Zyklus  $v_1, \dots, v_k$ . Für  $l \in \{k, \dots, |V|\}$  definieren wir einen Teilgraphen  $G_l = (V_l, E_l)$  von  $G$  mit  $|V_l| = |E_l| = l$  wie folgt: Falls  $l = k$ , dann besteht  $G_k$  aus dem Zyklus  $v_1, \dots, v_k$ . Sei  $l > k$ . Da  $G$  zusammenhängend ist, existiert ein  $\{v, w\} \in E$  so dass  $v \in V_{l-1}$  und  $w \notin V_{l-1}$ . Wir definieren  $V_l = V_{l-1} \cup \{w\}$  und  $E_l = E_{l-1} \cup \{\{v, w\}\}$ . Dann gilt  $|V_l| = |E_l| = l$ . Für  $l = |V|$  erhalten wir also einen Teilgraphen  $(V, E_{|V|})$  von  $(V, E)$ . Damit ist  $|E_V| = |V| \leq |E|$  was aber  $|E| = |V| - 1$  widerspricht.

5.  $\Rightarrow$  6.: Angenommen es gäbe  $E' \supset E$  so dass  $(V, E')$  zyklensfrei wäre, dann wäre wegen Lemma 2.2 ja  $|E'| \leq |V| - 1$  was  $|E'| > |E| = |V| - 1$  widerspricht.

6.  $\Rightarrow$  1.: Es reicht zu zeigen dass  $G$  zusammenhängend ist. Angenommen  $G$  wäre nicht zusammenhängend. Seien dann  $v$  und  $w$  Knoten aus verschiedenen Zusammenhangskomponenten und  $G' = (V, E \cup \{\{v, w\}\})$ . Dann ist  $G'$  immer noch zyklensfrei, denn jeder Zyklus in  $G'$  wäre entweder Zyklus in  $G$  (Widerspruch) oder er würde die Kante  $\{v, w\}$  mindestens zwei Mal enthalten woraus sich ein Zyklus in jeder der beiden Zusammenhangskomponenten bilden ließe (Widerspruch).  $G$  ist also nicht maximal zyklensfrei, Widerspruch.  $\square$

**Korollar 2.1.** *Ein endlicher Wald  $(V, E)$  besteht aus  $|V| - |E|$  Bäumen.*

*Beweis.* Angenommen  $(V, E)$  besteht aus den  $m$  Bäumen  $(V_1, E_1), \dots, (V_m, E_m)$ . Dann ist

$$|E| = \sum_{i=1}^m |E_i| \stackrel{\text{Satz 2.3}}{=} \sum_{i=1}^m (|V_i| - 1) = \sum_{i=1}^m |V_i| - m = |V| - m$$

und damit  $m = |V| - |E|$ .  $\square$

Wir kommen jetzt zu einer Anwendung von Bäumen. Angenommen wir wollen eine Menge  $V$  von Orten (z.B. Computern in einem Gebäude oder Pins auf einer Leiterplatte) verbinden. Die Knoten  $V$  bilden gemeinsam mit möglichen Verbindungen  $E \subseteq V \times V$  einen ungerichteten Graphen. Zusätzlich sind die Kosten des Legens einer Verbindung bekannt, d.h. eine Kostenfunktion  $c : E \rightarrow \mathbb{R}_{\geq 0}$  ist gegeben. Der Graph der Verbindungen soll also  $G' = (V, E')$  sein wobei  $E' \subseteq E$  ist,  $G'$  zusammenhängend ist und  $\sum_{e \in E'} c(e)$  minimal sein soll.

**Korollar 2.2.** *Sei  $G = (V, E)$  ein zusammenhängender Graph und  $E' \subseteq E$  so dass  $G' = (V, E')$  minimal zusammenhängend ist. Dann ist  $G'$  ein Baum.*

*Beweis.* Folgt unmittelbar aus Satz 2.3.  $\square$

Der gesuchte Verbindungsgraph ist also ein Baum. Das motiviert die folgenden Definitionen und das folgende Berechnungsproblem.

**Definition 2.9.** Sei  $G = (V, E)$  ein zusammenhängender Graph. Ein *Spannbaum* von  $G$  ist ein Baum  $B = (V, E')$  mit  $E' \subseteq E$ .

Der Baum  $B$  spannt also  $G$  auf.

**Definition 2.10.** Sei  $G = (V, E)$  ein zusammenhängender Graph und sei  $c : E \rightarrow \mathbb{R}_{\geq 0}$ . Ein *minimaler Spannbaum* von  $G$  bezüglich  $c$  ist ein Spannbaum  $B = (V, E')$  von  $G$  so dass  $\sum_{e \in E'} c(e)$  minimal ist unter aller Spannbäumen von  $G$ .

### Minimaler Spannbaum

Eingabe: Ein zusammenhängender Graph  $G$  und eine Kostenfunktion

$$c : E \rightarrow \mathbb{R}_{\geq 0}$$

Ausgabe: Ein minimaler Spannbaum von  $G$  bezüglich  $c$

Wir werden später effiziente Algorithmen zur Bestimmung eines minimalen Spannbaums kennen lernen.

# Kapitel 3

## Teile und Herrsche

In der Praxis auftretende Berechnungsprobleme haben oft die Eigenschaft dass ihre Instanzen zerlegt werden können und Lösungen der kleineren Instanzen zur Lösung der ursprünglichen Instanz hilfreich sind. Diese Vorgehensweise zur Lösung eines Berechnungsproblems ist so weit verbreitet, dass sich dafür ein eigener Begriff eingebürgert hat: die *teile-und-herrsche Strategie*. Algorithmen, die der teile-und-herrsche Strategie folgen bestehen üblicherweise aus drei Phasen:

1. *Aufteilung* der Eingabeinstanz in mehrere Instanzen kleinerer Größe
2. *Lösung* der kleineren Instanzen durch rekursiven Aufruf des Algorithmus
3. *Kombination* der Lösungen der kleineren Instanzen zu einer Lösung der ursprünglichen Instanz

Die Basis dieser Rekursion wird normalerweise dadurch gebildet, dass Instanzen deren Größe eine Konstante ist trivial gelöst werden können.

### 3.1 Sortieren durch Verschmelzen

Das Sortierproblem kann durch einen teile-und-herrsche Algorithmus gelöst werden. Die Grundidee dazu ist 1. das Eingabedatenfeld in zwei (zirka) gleich große Teile zu teilen, 2. jeden dieser beiden Teile unabhängig vom anderen durch einen rekursiven Aufruf zu sortieren und 3. die beiden sortierten Datenfelder zu einem einzigen sortierten Datenfeld zu verschmelzen. Deshalb wird dieser Algorithmus auch als “Sortieren durch Verschmelzen” (engl. *merge sort*) bezeichnet. Von den beiden Schritten 1 und 2 sollte klar sein wie sie umgesetzt werden können. Der Ansatz zur Realisierung des 3. Schritts besteht darin zwei Indizes zu verwenden, jeweils einen für jedes der Eingabedatenfelder, und sie so durch die Eingabedatenfelder laufen zu lassen, dass das aktuelle Minimum immer unter einem der beiden steht. Dieses aktuelle Minimum wird in das Ausgabedatenfeld übertragen. In Algorithmus 3 ist diese Idee als Pseudocode ausformuliert.

Basierend auf der Prozedur Verschmelzen kann nun Sortieren durch Verschmelzen wie in Algorithmus 4 als Pseudocode formuliert werden. Für  $i \leq j$  ist die Notation  $A := B[i, \dots, j]$  eine Abkürzung für die Herstellung eines Datenfeldes  $A$  dessen Inhalt genau den Einträgen  $i$  bis  $j$  des Datenfeldes  $B$  entspricht. Eine solche Kopie kann durch eine einfache Schleife in Laufzeit  $\Theta(j - i)$  hergestellt werden.

Wir wollen nun die Laufzeit von Sortieren durch Verschmelzen analysieren. Die Prozedur “Verschmelzen” benötigt Laufzeit  $\Theta(A.Länge + B.Länge)$ . Sei  $T(n)$  die Laufzeit von Sortieren durch

---

**Algorithmus 3** Verschmelzen

---

**Prozedur** VERSCHMELZEN( $A, B$ )

Sei  $C$  ein neues Datenfeld der Länge  $A.Länge + B.Länge$

$i := 1$

$j := 1$

**Für**  $k := 1, \dots, A.Länge + B.Länge$

**Falls**  $j > B.Länge$  **oder** ( $i \leq A.Länge$  **und**  $A[i] \leq B[j]$ ) **dann**

$C[k] := A[i]$

$i := i + 1$

**sonst**

$C[k] := B[j]$

$j := j + 1$

**Ende Falls**

**Ende Für**

Antworte  $C$

**Ende Prozedur**

---

---

**Algorithmus 4** Sortieren durch Verschmelzen (engl. *merge sort*)

---

**Prozedur** VSORTIEREN( $A$ )

**Falls**  $A.Länge = 1$  **dann**

Antworte  $A$

**sonst**

$m := \left\lceil \frac{A.Länge}{2} \right\rceil$

$L := A[1, \dots, m]$

$R := A[m + 1, \dots, A.Länge]$

$L' := \text{VSORTIEREN}(L)$

$R' := \text{VSORTIEREN}(R)$

Antworte  $\text{VERSCHMELZEN}(L', R')$

**Ende Falls**

**Ende Prozedur**

---

Verschmelzen wenn das Eingabedatenfeld die Länge  $n$  hat. Klar ist bereits dass  $T(1) = \Theta(1)$ . Für  $n \geq 2$  beobachten wir: Die Aufteilung in Teilprobleme benötigt Laufzeit  $\Theta(n)$ , das Lösen der Teilprobleme benötigt  $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$ , das Verschmelzen der beiden Lösungen, die ja Größe  $m$  und  $n - m$  haben, benötigt  $\Theta(n)$ . Insbesondere können wir hier beobachten, dass die Laufzeit von Sortieren durch Verschmelzen, anders als die von Einfügesortieren, nur von  $n = A.Länge$  abhängt, nicht aber vom Inhalt von  $A$ . Somit ist die Laufzeit im besten Fall gleich der Laufzeit im schlechtesten Fall und gleich der Laufzeit im Durchschnittsfall.

Insgesamt erhalten wir für die Laufzeit

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}$$

Bei dieser Darstellung der Laufzeit handelt es sich um eine *Rekursionsgleichung*. Die Analyse eines teile-und-herrsche Algorithmus führt typischerweise auf eine Rekursionsgleichung einer solchen Form. In Kapitel 4 werden wir zeigen, dass für die obige Rekursionsgleichung  $T(n) = \Theta(n \log n)$  gilt<sup>1</sup>. Für den Augenblick wollen wir uns damit begnügen die folgende einfache Beobachtung zu machen, die den Kern der obigen Rekursionsgleichung enthält.

**Satz 3.1.** Sei  $S : \{2^k \mid k \geq 1\} \rightarrow \mathbb{N}$  definiert durch  $S(n) = \begin{cases} 2 & \text{falls } n = 2 \\ 2S(\frac{n}{2}) + n & \text{falls } n > 2 \end{cases}$ , dann ist  $S(n) = n \log n$ .

*Beweis.* Zu zeigen ist also, für alle  $k \geq 1$ , dass  $S(2^k) = k \cdot 2^k$ . Für  $k = 1$  ist das per definitionem erfüllt. Für  $k \geq 2$  haben wir  $S(2^k) = 2 \cdot S(2^{k-1}) + 2^k \stackrel{\text{IH}}{=} 2 \cdot (k-1) \cdot 2^{k-1} + 2^k = k \cdot 2^k$ .  $\square$

## 3.2 Matrixmultiplikation

Wir werden annehmen, dass eine Matrix im Speicher abgelegt wird als ein Datenfeld, dessen Elemente Datenfelder einer festen Länge sind. Falls also  $A$  eine Matrix ist, so ist  $A[i]$  die  $i$ -te Zeile von  $A$  und damit  $A[i][j]$  das  $j$ -te Element der  $i$ -ten Zeile. Um die Notation etwas abzukürzen, schreiben wir stattdessen auch  $A[i, j]$ . Wir betrachten das folgende Problem:

**Matrixmultiplikation**  
 Eingabe: eine  $n \times m$ -Matrix  $A$  und eine  $m \times l$ -Matrix  $B$   
 Ausgabe:  $A \cdot B$

Seien  $n, m, l \geq 1$  und  $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}}$  und  $B = (b_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq l}}$  Matrizen. Dann ist deren Produkt  $A \cdot B = C = (c_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq l}}$  bekannterweise durch

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

definiert. Diese Formel induziert sofort Algorithmus 5. Dessen Laufzeit kann aufgrund der drei verschachtelten Schleifen sofort als  $\Theta(n \cdot l \cdot m)$  erkannt werden.

<sup>1</sup>In dieser Vorlesung werden wir mit  $\log$  immer den Logarithmus zur Basis 2 notieren

---

**Algorithmus 5** Matrixmultiplikation (direkt)

---

**Vorbedingung:**  $n, m, l \geq 1$ ,  $A$  ist  $n \times m$ -Matrix,  $B$  ist  $m \times l$ -Matrix

**Prozedur** MATMULT( $A, B$ )

Sei  $C$  eine neue  $n \times l$ -Matrix

**Für**  $i := 1, \dots, n$

**Für**  $j := 1, \dots, l$

$C[i, j] := 0$

**Für**  $k := 1, \dots, m$

$C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$

**Ende Für**

**Ende Für**

**Ende Für**

**Antworte**  $C$

**Ende Prozedur**

---

Um die Analyse zu vereinfachen, werden wir von nun an annehmen, dass  $n = m = l$  und dass  $n$  eine Zweierpotenz ist. Für  $n = 2n'$  kann eine  $n \times n$ -Matrix in vier  $n' \times n'$ -Matrizen geteilt werden.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Damit kann eine  $n \times n$ -Matrix mit Elementen aus einer Menge  $R$  identifiziert werden mit einer  $2 \times 2$ -Matrix deren Elemente  $n' \times n'$ -Matrizen mit Elementen aus  $R$  sind (Diese Beobachtung kann präzisiert werden, z.B. für einen Ring  $R$ , indem man zeigt, dass die Matrizenringe  $R^{n \times n}$  und  $(R^{n' \times n'})^{2 \times 2}$  isomorph sind). Für  $C = A \cdot B$  erhalten wir damit die Darstellung

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

der  $n \times n$ -Multiplikation durch  $n' \times n'$ -Multiplikation. Diese Beobachtung induziert das in Algorithmus 6 angegebene einfache teile-und-herrsche Verfahren. Hier schreiben wir im Sinne der soeben diskutierten Teilung  $A_{1,1}$  für  $A[1, \dots, n'; 1, \dots, n']$ ,  $A_{1,2}$  für  $A[1, \dots, n'; n' + 1, \dots, n]$ , usw. auch für  $B$  und  $C$ . Die Laufzeit dieses Algorithmus erfüllt also:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 8T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n \geq 2 \end{cases}$$

Diese Rekursionsgleichung hat die asymptotische Lösung  $T(n) = \Theta(n^3)$  wie wir im Kapitel 4 sehen werden. Dieser teile-und-herrsche Algorithmus ist also asymptotisch nicht effizienter als das direkte Verfahren. Man könnte nun glauben, dass der teile-und-herrsche-Ansatz für die Matrixmultiplikation keine Verbesserung des direkten Verfahren erlaubt, oder sogar dass ein Algorithmus mit einer Laufzeit von weniger als  $\Theta(n^3)$  gar nicht möglich ist, da dies ja der natürlichen Definition des Produkts entspricht. Beides wäre allerdings ein Irrtum wie der Algorithmus von Strassen zeigt.

Der Algorithmus von Strassen folgt ebenso wie Algorithmus 6 der teile-und-herrsche-Methode. Er unterscheidet sich von ihm dadurch, dass er eine geschicktere Darstellung der  $C_{i,j}$  findet, die

---

**Algorithmus 6** Matrixmultiplikation (rekursiv)

---

**Vorbedingung:**  $A, B$  sind  $n \times n$  Matrizen,  $n$  ist Zweierpotenz

**Prozedur** MATMULT( $A, B$ )

Sei  $C$  eine neue  $n \times n$ -Matrix

**Falls**  $n = 1$  **dann**

$$C[1, 1] := A[1, 1] \cdot B[1, 1]$$

**sonst**

$$C_{1,1} := \text{MATMULT}(A_{1,1}, B_{1,1}) + \text{MATMULT}(A_{1,2}, B_{2,1})$$

$$C_{1,2} := \text{MATMULT}(A_{1,1}, B_{1,2}) + \text{MATMULT}(A_{1,2}, B_{2,2})$$

$$C_{2,1} := \text{MATMULT}(A_{2,1}, B_{1,1}) + \text{MATMULT}(A_{2,2}, B_{2,1})$$

$$C_{2,2} := \text{MATMULT}(A_{2,1}, B_{1,2}) + \text{MATMULT}(A_{2,2}, B_{2,2})$$

**Ende Falls**

**Antworte**  $C$

**Ende Prozedur**

---

mit sieben Multiplikationen (von  $n' \times n'$ -Matrizen) auskommt. Seien nämlich

$$\begin{aligned} P_1 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}), \\ P_2 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2}, \\ P_3 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1}, \\ P_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}), \\ P_5 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}), \\ P_6 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}), \text{ und} \\ P_7 &= (A_{1,1} - A_{2,1}) \cdot (B_{1,1} + B_{1,2}). \end{aligned}$$

Dann gilt

$$\begin{aligned} C_{1,1} &= P_5 + P_4 - P_2 + P_6, \\ C_{1,2} &= P_1 + P_2, \\ C_{2,1} &= P_3 + P_4, \text{ und} \\ C_{2,2} &= P_5 + P_1 - P_3 - P_7, \end{aligned}$$

wovon man sich durch eine kurze Rechnung überzeugen kann. Der Strassen-Algorithmus ist dann wie folgt wobei wir (wie oben) die Abkürzungen  $A_{i,j}$ ,  $B_{i,j}$  und  $C_{i,j}$  auch im Pseudocode verwenden.

---

**Algorithmus 7** Strassen-Algorithmus

---

**Vorbedingung:**  $A, B$  sind  $n \times n$  Matrizen,  $n$  ist Zweierpotenz

**Prozedur** STRASSEN( $A, B$ )

Sei  $C$  eine neue  $n \times n$ -Matrix

**Falls**  $n = 1$  **dann**

$$C[1, 1] := A[1, 1] \cdot B[1, 1]$$

**sonst**

$$P_1 := \text{STRASSEN}(A_{1,1}, B_{1,2} - B_{2,2})$$

$$P_2 := \text{STRASSEN}(A_{1,1} + A_{1,2}, B_{2,2})$$

$$P_3 := \text{STRASSEN}(A_{2,1} + A_{2,2}, B_{1,1})$$

$$P_4 := \text{STRASSEN}(A_{2,2}, B_{2,1} - B_{1,1})$$

$$P_5 := \text{STRASSEN}(A_{1,1} + A_{2,2}, B_{1,1} + B_{2,2})$$

$$P_6 := \text{STRASSEN}(A_{1,2} - A_{2,2}, B_{2,1} + B_{2,2})$$

$$P_7 := \text{STRASSEN}(A_{1,1} - A_{2,1}, B_{1,1} + B_{1,2})$$

$$C_{1,1} := P_5 + P_4 - P_2 + P_6$$

$$C_{1,2} := P_1 + P_2$$

$$C_{2,1} := P_3 + P_4$$

$$C_{2,2} := P_5 + P_1 - P_3 - P_7$$

**Ende Falls**

**Antworte**  $C$

**Ende Prozedur**

---

Die Laufzeit vom Strassen-Algorithmus erfüllt also:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n > 1 \end{cases}$$

Im nächsten Kapitel werden wir zeigen dass die asymptotische Lösung dieser Rekursionsgleichung  $T(n) = \Theta(n^{\log 7})$  ist. Da  $\log 7 \approx 2,807$  wird dadurch eine Verbesserung des direkten sowie des rekursiven Verfahrens erreicht. Da die Konstanten des Strassen-Algorithmus aber größer sind als bei den beiden anderen Verfahren, wird in der Praxis meist eine Kombination verwendet: für hinreichend große Matrizen wird der Strassen-Algorithmus eingesetzt, für kleinere ein direkteres Verfahren.

Der Algorithmus von Strassen ist ein gutes Beispiel für einen teile-und-herrsche Algorithmus mit einer trickreichen Aufteilungs- und damit auch Kombinationsphase.

*Bemerkung 3.1.* Die Einschränkung auf quadratische Matrizen und  $n$  einer Zweierpotenz ist weder für den einfachen teile-und-herrsche Algorithmus, noch für den Strassen-Algorithmus notwendig. Diese Einschränkung dient lediglich der einfacheren Darstellung da durch sie einige Sonderfälle nicht behandelt werden müssen. In der Tat kann für beliebige  $n, m, l$  die Multiplikation einer  $n \times m$ - mit einer  $m \times l$ -Matrix durch Multiplikationen von Matrizen kleinerer Größe dargestellt werden, durch eine Zerlegung der Form  $n = n_1 + n_2$ ,  $m = m_1 + m_2$ ,  $l = l_1 + l_2$ . Setzt man zum Beispiel für  $x \in \{n, m, l\}$  jeweils  $x_1 = \lceil \frac{x}{2} \rceil$  und  $x_2 = \lfloor \frac{x}{2} \rfloor$  erhält man einen allgemeineren teile-und-herrsche Algorithmus. Ähnliches gilt für den Strassen-Algorithmus. Für diesen ist allerdings notwendig dass alle  $A_{i,j}$  die selbe Größe haben und dass alle  $B_{i,j}$  die selbe Größe haben, d.h. also dass  $n, m$  und  $l$  gerade sind. Das kann erreicht werden durch Anfügen einer Nullzeile oder Nullspalte im Bedarfsfall.

*Bemerkung 3.2.* Der Algorithmus von Strassen wurde im Jahr 1969 publiziert und hat, wie beschrieben, eine Laufzeitkomplexität von  $O(n^{2,807\dots})$ . Seitdem konnten weitere Verbesserungen der asymptotischen Laufzeit der Matrixmultiplikation erreicht werden. Ein signifikanter

Schritt vorwärts war der Algorithmus von Coppersmith-Winograd aus dem Jahr 1990 mit einer Laufzeitkomplexität von  $O(n^{2,376})$ . Über diesen hinaus konnten bis heute nur geringe Verbesserungen erreicht werden, so z.B.  $O(n^{2,374})$  von Stothers 2010 und  $O(n^{2,373})$  von Williams 2011. Diese weiteren Algorithmen haben allerdings so große Konstanten, dass sie in der Praxis keine Bedeutung haben.

Untere Schranken zur Matrixmultiplikation sind kaum bekannt. Klar ist, dass  $\Omega(n^2)$  eine triviale untere Schranke ist, da die Eingabe der Größe  $2n^2$  ja gelesen und die Ausgabe der Größe  $n^2$  geschrieben werden muss. Eine untere Schranke von  $\Omega(n^2 \log n)$  auf der Größe einer gewissen, eingeschränkten, Klasse von Schaltkreisen wurde von Raz 2003 bewiesen.

### 3.3 Dichtestes Punktepaar

Wir betrachten jetzt ein erstes fundamentales geometrisches Berechnungsproblem. Für Punkte  $p_1, p_2 \in \mathbb{R}^2$  sei  $d(p_1, p_2)$  die übliche euklidische Distanz, d.h.

$$d\left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Wir wollen das folgende Problem lösen:

**Dichtestes Punktepaar**

Eingabe: Eine endliche Menge  $P \subseteq \mathbb{R}^2$

Ausgabe:  $p, q \in P$  so dass  $p \neq q$  und  $d(p, q)$  minimal

Klar ist, dass ein auf erschöpfender Suche basierender Algorithmus existiert, der einfach alle Paare von Punkten ausprobiert, siehe Algorithmus 8. Es gibt  $\binom{n}{2}$  Paare die alle in jedem Fall

---

**Algorithmus 8** Erschöpfende Suche nach dichtestem Punktepaar

---

**Prozedur** DPP-SUCHE( $P$ )

$d_{\min} := \infty$

**Für**  $i := 1, \dots, P.Länge$

**Für**  $j := i + 1, \dots, P.Länge$

**Falls**  $d(P[i], P[j]) < d_{\min}$  **dann**

$d_{\min} := d(P[i], P[j])$

$a := (P[i], P[j])$

**Ende Falls**

**Ende Für**

**Ende Für**

**Antwort**  $a$

**Ende Prozedur**

---

durchlaufen werden, damit hat dieser Algorithmus Laufzeit  $\Theta(n^2)$ .

Wir werden sehen, dass es möglich ist mit einem Algorithmus der dem teile-und-herrsche Prinzip folgt (selbst im schlechtesten Fall) eine Laufzeit von  $O(n \log n)$  zu erreichen.

Sei  $P \subseteq \mathbb{R}^2$  eine endliche Menge von Punkten. Die Grundidee des Verfahrens ist wie folgt:

1. *Aufteilung*: Wir teilen das Problem entlang einer vertikalen Gerade, genauer: Sei  $x_m \in \mathbb{R}$  und  $P = Q \uplus R$  so dass  $|Q| = \lceil \frac{|P|}{2} \rceil$ ,  $|R| = \lfloor \frac{|P|}{2} \rfloor$ , für alle  $\begin{pmatrix} x \\ y \end{pmatrix} \in Q$ :  $x \leq x_m$  und für alle  $\begin{pmatrix} x \\ y \end{pmatrix} \in R$ :  $x \geq x_m$ . Man beachte, dass sowohl  $P$  als auch  $Q$  Punkte mit  $x$ -Koordinate  $x_m$  enthalten können.

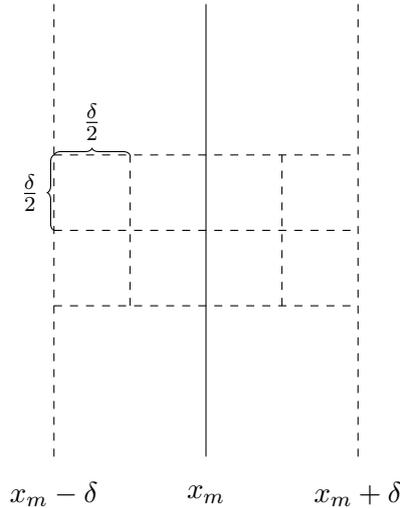


Abbildung 3.1: Kombination der Lösungen der Teilprobleme

2. *Lösung:* Mit Hilfe rekursiver Aufrufe bestimmen wir das dichteste Punktepaar  $q_1, q_2 \in Q$  sowie das dichteste Punktepaar  $r_1, r_2 \in R$ . Sei  $\delta = \min\{d(q_1, q_2), d(r_1, r_2)\}$ .

3. *Kombination:* Das dichteste Punktepaar von  $P$  ist nun entweder  $q_1, q_2$  oder  $r_1, r_2$  oder ein Paar  $q, r$  wobei  $q \in Q$  und  $r \in R$ . Natürlich können wir jetzt nicht alle (quadratisch vielen) Paare in  $Q \times R$  durchsuchen wenn wir nur Laufzeit  $O(n \log n)$  verwenden wollen. Mit einer kurzen Überlegung kann man aber zeigen, dass es ausreicht linear viele Paare zu überprüfen.

Der Schlüssel dazu ist, dass wir  $\delta$  bereits kennen. Falls nämlich ein Paar  $q = \begin{pmatrix} q_x \\ q_y \end{pmatrix} \in Q$ ,

$r = \begin{pmatrix} r_x \\ r_y \end{pmatrix} \in R$  das dichteste Punktepaar von  $P$  ist, muss nämlich  $q_x, r_x \in [x_m - \delta, x_m + \delta]$  sein,

d.h. dass  $q$  und  $r$  in einem Schlauch der Breite  $2\delta$  um  $x_m$  liegen. Weiters muss natürlich auch  $|q_y - r_y| < \delta$  sein. Also liegen  $q$  und  $r$  in einem  $2\delta \times \delta$  großen Rechteck das um die Gerade  $x = x_m$  zentriert ist. Dieses Rechteck stellen wir uns nun als unterteilt in 8 Zellen der Größe  $\frac{\delta}{2} \times \frac{\delta}{2}$  vor, siehe Abbildung 3.1. Jeder der Zellen (als Produkt geschlossener Intervalle) auf der linken Seite (von  $x = x_m$ ) enthält höchstens einen Punkt von  $Q$ : würde eine Zelle nämlich zwei Punkte

$q'_1, q'_2 \in Q$  enthalten, dann wäre  $d(q'_1, q'_2) \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$  was ein Widerspruch ist.

Analog dazu enthält auch jede Zelle auf der rechten Seite höchstens einen Punkt von  $R$ . In diesem Rechteck gibt es also höchstens 8 Punkte von  $P$ . Um alle für das dichteste in Frage kommenden Punktepaare  $q \in Q, r \in R$  zu überprüfen reicht es also, die Punkte im Schlauch rund um die Gerade  $x = x_m$  nach  $y$ -Koordinate zu sortieren und für jeden Punkt den Abstand zu seinen 7 nächsten in der sortierten Liste zu überprüfen. Falls auf diese Weise ein Punktepaar  $q \in Q, r \in R$  mit  $d(q, r) < \delta$  gefunden wird, so ist dieses das dichteste in  $P$ , falls nicht, dann ist das dichteste Punktepaar in  $P$  je nachdem entweder  $q_1, q_2$  oder  $r_1, r_2$ . Damit haben wir uns also geometrisch davon überzeugt, dass diese Idee für einen teile-und-herrsche-Algorithmus sinnvoll ist und wir können uns an die konkrete Realisierung machen.

Ein Aspekt der durch die obige Diskussion noch nicht vollständig festgelegt ist, ist was passieren soll wenn mehrere Punkte in  $P$  auf der Gerade  $x = x_m$  liegen. In solchen nicht-deterministischen Situationen ist es häufig nützlich, eine einfach zu berechnende Determinisierung festzulegen. Dazu definieren wir:

**Definition 3.1.** Die *lexikographische Ordnung*  $<_{\text{lex}}$  auf  $\mathbb{R}^2$  ist festgelegt durch:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} <_{\text{lex}} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \text{ genau dann wenn } 1. x_1 < x_2 \text{ oder} \\ 2. x_1 = x_2 \text{ und } y_1 < y_2.$$

Wir schreiben  $\leq_{\text{lex}}$  für die reflexive Hülle von  $<_{\text{lex}}$  und beobachten, dass  $<_{\text{lex}}$  eine totale Ordnung auf  $\mathbb{R}^2$  ist. Wir können die Aufteilung des Problems für die Menge  $P$  nun also deterministisch spezifizieren als: Sei  $p_m \in P$ ,  $Q = \{p \in P \mid p \leq_{\text{lex}} p_m\}$ ,  $R = \{p \in P \mid p >_{\text{lex}} p_m\}$  so dass  $|Q| = \lceil \frac{|P|}{2} \rceil$  und  $R = \lfloor \frac{|P|}{2} \rfloor$ . Dann ist  $x_m$  die  $x$ -Koordinate von  $p_m$ . Falls also mehrere Punkte in  $P$  die  $x$ -Koordinate  $x_m$  haben gibt es auf der Gerade  $x = x_m$  einen Punkt so dass alle darunter liegenden Punkte in  $Q$  sind und alle darüber liegenden in  $R$ .

Wir benötigen also eine Darstellung der Punkte sortiert nach  $\leq_{\text{lex}}$  sowie eine in der die selben Punkte nach  $y$ -Koordinate sortiert sind. Das können wir erreichen, indem wir zwei Datenfelder verwenden die die selben Punkte enthalten: eines wird nach  $\leq_{\text{lex}}$  sortiert und eines nach der  $y$ -Koordinate. Ein wichtiger Punkt ist nun dass wir es uns nicht leisten können, nach der Teilung erneut zu sortieren. Das würde in jedem Schritt Zeit  $O(n \log n)$  verbrauchen und um eine Gesamtlaufzeit von  $O(n \log n)$  zu erreichen müssen wir in jedem Schritt mit Zeit  $O(n)$  auskommen. Dieses Hindernis kann überwunden werden, indem wir diese beiden Sortierungen einmal zu Beginn des Algorithmus (in Zeit  $O(n \log n)$ ) berechnen und dann sicherstellen, dass die Sortierung durch den gesamten Algorithmus hindurch beibehalten wird.

Zur Realisierung dieses Ansatzes benötigen wir noch eine weitere Operation. Falls wir aus einer Menge  $X$  alle Elemente  $x$  auswählen wollen, die eine Eigenschaft  $P(x)$  haben, dann schreiben wir diese Menge als  $\{x \in X \mid P(x)\}$ . Eine analoge Operation auf Datenfeldern kann wie folgt definiert werden: Sei  $A$  ein Datenfeld, dann ist  $\langle x \in A \mid P(x) \rangle$  ein Datenfeld das alle Elemente von  $A$  enthält, die die Eigenschaft  $P$  erfüllen und zwar *in der Reihenfolge, in der sie in  $A$  vorkommen*. Algorithmus 9 führt diese Auswahl durch. Falls die Überprüfung der Eigenschaft

---

**Algorithmus 9** Auswahl aus einem Datenfeld

---

**Prozedur** AUSWAHL $_P(A)$   
 Sei  $B$  ein neues Datenfeld  
 $j := 1$   
**Für**  $i = 1, \dots, A.\text{Länge}$   
   **Falls**  $P(A[i])$  **dann**  
      $B[j] := A[i]$   
      $j := j + 1$   
**Ende Falls**  
**Ende Für**  
**Antworte**  $B$   
**Ende Prozedur**

---

$P$  konstante Zeit erfordert (was üblicherweise bei uns der Fall sein wird), dann läuft dieser Algorithmus in Zeit  $\Theta(n)$  wobei  $n = A.\text{Länge}$ .

Der gesamte Algorithmus zur Bestimmung des dichtesten Punktepaars kann als Pseudocode wie in Algorithmus 10 geschrieben werden.

Für die Laufzeitanalyse von DPP-TH sei  $n = |P|$ . Die Laufzeit von DPP-TH ist  $\Theta(n \log n)$  für das Sortieren plus der Laufzeit von DPP-TH-Rek. Für die Laufzeitkomplexität von DPP-TH-

---

**Algorithmus 10** Teile-und-herrsche Algorithmus für dichtestes Punktepaar

---

**Prozedur** DPP-TH( $P$ )

$P := P$  aufsteigend nach  $\leq_{\text{lex}}$  sortiert

$P_y := P$  aufsteigend nach  $y$ -Koordinate sortiert

**Antworte** DPP-TH-REK( $P, P_y$ )

**Ende Prozedur**

**Prozedur** DPP-TH-REK( $P, P_y$ )

**Falls**  $P.\text{Länge} \leq 3$  **dann**

**Antworte** DPP-SUCHE( $P$ )

**Ende Falls**

$m := \left\lceil \frac{P.\text{Länge}}{2} \right\rceil$

$Q := P[1, \dots, m]$

$R := P[m + 1, \dots, P_x.\text{Länge}]$

$Q_y := \langle p \in P_y \mid p \leq_{\text{lex}} P[m] \rangle$

$R_y := \langle p \in P_y \mid p >_{\text{lex}} P[m] \rangle$

$(q_1, q_2) := \text{DPP-TH-REK}(Q, Q_y)$

$(r_1, r_2) := \text{DPP-TH-REK}(R, R_y)$

$\delta := \min\{d(q_1, q_2), d(r_1, r_2)\}$

$S_y := \langle p \in P_y \mid P[m].x - \delta \leq p.x \leq P[m].x + \delta \rangle$

$s_1 := (0, 0)$

$s_2 := (\infty, \infty)$

**Für**  $i := 1, \dots, S_y.\text{Länge} - 1$

**Für**  $j := i + 1, \dots, \min\{i + 7, S_y.\text{Länge}\}$

**Falls**  $d(S_y[i], S_y[j]) < d(s_1, s_2)$  **dann**

$s_1 := S_y[i]$

$s_2 := S_y[j]$

**Ende Falls**

**Ende Für**

**Ende Für**

**Falls**  $d(s_1, s_2) < \delta$  **dann**

**Antworte**  $(s_1, s_2)$

**sonst falls**  $d(r_1, r_2) < d(q_1, q_2)$  **dann**

**Antworte**  $(r_1, r_2)$

**sonst**

**Antworte**  $(q_1, q_2)$

**Ende Falls**

**Ende Prozedur**

---

Rek erhalten wir

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 3 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 3 \end{cases}$$

Diese Rekursionsgleichung ist beinahe identisch zu jener die wir aus Sortieren durch Verschmelzen erhalten haben. In der Tat gilt auch hier  $T(n) = \Theta(n \log n)$  wie wir in Kapitel 4 zeigen werden. Somit ist auch die Laufzeit des Gesamtalgorithmus DPP-TH  $\Theta(n \log n)$ .

Wir sehen also, dass die Kombinationsphase in einem teile-und-herrsche-Algorithmus durchaus auch trickreicher sein kann als das beim Sortieren durch Verschmelzen der Fall ist.



# Kapitel 4

## Rekursionsgleichungen

Eine Rekursionsgleichung definiert eine Folge  $(x_n)_{n \geq 0}$  durch eine Gleichung, die  $x_n$  basierend auf  $n$  und den  $x_i$  mit  $i < n$  definiert sowie hinreichend vielen Anfangswerten. Typischerweise ist man daran interessiert, einen geschlossenen Ausdruck für  $x_n$  zu finden, d.h. einen, in dem keine  $x_i$  mehr vorkommen.

*Beispiel 4.1.* Die bekannte Rekursionsgleichung

$$F_n = F_{n-1} + F_{n-2} \text{ für } n \geq 2 \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$

definiert die Folge der Fibonacci-Zahlen  $0, 1, 1, 2, 3, 5, 8, 13, \dots$

### 4.1 Lineare Rekursionsgleichungen erster Ordnung

**Definition 4.1.** Die *Ordnung* einer Rekursionsgleichung ist das kleinste  $k$  so dass die Definition von  $x_n$  von  $\{x_{n-1}, \dots, x_{n-k}\}$  abhängt.

So hat zum Beispiel die Fibonacci-Gleichung die Ordnung 2.

**Definition 4.2.** Eine *lineare Rekursionsgleichung erster Ordnung* ist eine Rekursionsgleichung der Form

$$x_n = a_n x_{n-1} + b_n \text{ für } n \geq 1$$

mit festgelegten Anfangswert  $x_0$ .

*Beispiel 4.2.* Die Rekursionsgleichung

$$x_n = x_{n-1} + b_n \text{ für } n \geq 1 \text{ mit } x_0 = 0$$

hat als Lösung  $x_n = \sum_{i=1}^n b_i$ .

Die Rekursionsgleichung

$$x_n = a_n x_{n-1} \text{ für } n \geq 1 \text{ mit } x_0 = 1$$

hat als Lösung  $x_n = \prod_{i=1}^n a_i$ .

Diese beiden Beispiele können zu dem folgenden Resultat verallgemeinert werden:

**Satz 4.1.** Die Rekursionsgleichung  $x_n = a_n x_{n-1} + b_n$  für  $n \geq 1$  mit festgelegten Anfangswert  $x_0 = b_0$  hat die Lösung

$$x_n = \sum_{i=0}^n b_i \prod_{j=i+1}^n a_j.$$

*Beweis.* Mit Induktion: Für  $n = 0$  gilt  $x_0 = b_0$ . Für  $n > 0$  haben wir

$$x_n = a_n x_{n-1} + b_n = a_n \left( \sum_{i=0}^{n-1} b_i \prod_{j=i+1}^{n-1} a_j \right) + b_n = \left( \sum_{i=0}^{n-1} b_i \prod_{j=i+1}^n a_j \right) + b_n = \sum_{i=0}^n b_i \prod_{j=i+1}^n a_j.$$

□

## 4.2 Lineare Rekursionsgleichungen $k$ -ter Ordnung

Wir wollen nun lineare Rekursionsgleichungen beliebiger Ordnung betrachten, schränken uns dabei aber auf den Fall konstanter Koeffizienten ein.

**Definition 4.3.** Eine *homogene lineare Rekursionsgleichung mit konstanten Koeffizienten  $k$ -ter Ordnung* ist von der Form

$$x_n = c_{k-1}x_{n-1} + \dots + c_0x_{n-k} \text{ für } n \geq k \quad (4.1)$$

Falls die Konstanten  $c_0, \dots, c_{k-1}$  Elemente eines Körpers  $K$  sind, können wir die Rekursionsgleichung (4.1) über diesem Körper  $K$  auffassen. Wir sagen dass  $(a_n)_{n \geq 0}$  eine Lösung von (4.1) in  $K$  ist falls alle  $a_n \in K$  sind und  $a_n = c_{k-1}a_{n-1} + \dots + c_0a_{n-k}$  für alle  $n \geq k$ . Wir definieren  $K^\omega = \{(a_n)_{n \geq 0} \mid a_n \in K \text{ für alle } n \geq 0\}$  und stellen fest, dass  $K^\omega$  ein Vektorraum unendlicher Dimension über  $K$  ist.

**Lemma 4.1.** Sei  $K$  ein Körper, seien  $c_0, \dots, c_{k-1} \in K$ , dann ist die Lösungsmenge von (4.1) in  $K$  ein Untervektorraum von  $K^\omega$  mit Dimension  $k$ .

*Beweis.* Klar ist, dass die Lösungsmenge eine Teilmenge von  $K^\omega$  ist. Weiters ist die Lösungsmenge nicht leer, da z.B.  $(0)_{n \geq 0}$  eine Lösung von (4.1) ist. Seien nun  $(a_n)_{n \geq 0}, (b_n)_{n \geq 0}$  Lösungen von (4.1) und  $\lambda, \mu \in K$ , dann ist für  $n \geq k$

$$\begin{aligned} a_n &= c_{k-1}a_{n-1} + \dots + c_0a_{n-k}, \\ b_n &= c_{k-1}b_{n-1} + \dots + c_0b_{n-k}, \end{aligned}$$

und damit auch

$$\lambda a_n + \mu b_n = c_{k-1}(\lambda a_{n-1} + \mu b_{n-1}) + \dots + c_0(\lambda a_{n-k} + \mu b_{n-k}),$$

d.h. also auch  $(\lambda a_n + \mu b_n)_{n \geq 0}$  ist eine Lösung von (4.1) und damit ist die Lösungsmenge ein Unterraum von  $K^\omega$ . Seien  $a_0, \dots, a_{k-1}$  beliebig, dann ist  $(a_n)_{n \geq 0}$  eindeutig bestimmt, und kann geschrieben werden als  $(a_n)_{n \geq 0} = a_0 e_0 + \dots + a_{k-1} e_{k-1}$  wobei  $e_i \in K^\omega$  jene Lösung von (4.1) ist, die durch  $e_{i,j} = \delta_{i,j}$  für  $j = 0, \dots, k-1$  bestimmt wird. Also ist  $e_0, \dots, e_{k-1}$  Basis des Lösungsraums und damit ist seine Dimension  $k$ . □

Für den Spezialfall  $k = 1$  und  $x_0 = 1$  von (4.1) haben wir in Beispiel 4.2 bereits festgestellt, dass die Lösung  $x_n = c_0^n$  ist. Es liegt also nahe, Lösungen von der Form  $(\alpha^n)_{n \geq 0}$  zu betrachten. Für eine solche Lösung gilt

$$\alpha^n - c_{k-1}\alpha^{n-1} - \dots - c_0\alpha^{n-k} = 0.$$

für alle  $n \geq k$  und insbesondere für  $n = k$ :

$$\alpha^k - c_{k-1}\alpha^{k-1} - \dots - c_1\alpha - c_0 = 0.$$

Diese Beobachtung motiviert die folgende Definition:

**Definition 4.4.** Das *charakteristische Polynom* der Rekursionsgleichung (4.1) ist

$$\chi(z) = z^k - c_{k-1}z^{k-1} - \dots - c_1z - c_0$$

Das heißt also: falls  $(\alpha^n)_{n \geq 0}$  Lösung von (4.1) ist, dann ist  $\alpha$  Nullstelle von  $\chi(z)$ . Es gilt auch eine (stärkere) Implikation in die andere Richtung so dass wir das folgende Resultat erhalten:

**Satz 4.2.** Sei  $\chi(z)$  das charakteristische Polynom von (4.1) mit Nullstellen  $\alpha_1, \dots, \alpha_r$  und Vielfachheiten  $m_1, \dots, m_r$ . Dann ist  $\{(n^j \alpha_i^n)_{n \geq 0} \mid 1 \leq i \leq r, 0 \leq j < m_i\}$  eine Basis des Lösungsraums von (4.1).

*Beweis.* Sei  $\alpha$  eine Nullstelle von  $\chi(z)$ , dann ist

$$\alpha^k = c_{k-1}\alpha^{k-1} + \dots + c_1\alpha + c_0$$

und nach Multiplikation mit  $\alpha^{n-k}$

$$\alpha^n = c_{k-1}\alpha^{n-1} + \dots + c_1\alpha^{n-k+1} + c_0\alpha^{n-k}.$$

Somit ist  $(\alpha^n)_{n \geq 0}$  eine Lösung.

Sei  $\alpha$  Doppelnullestelle von  $\chi(z)$ , dann ist  $\alpha$  auch Nullstelle von  $\chi'(z)$  und damit vom Polynom  $\psi(z) = (n-k)\chi(z) + z\chi'(z)$ . Nun ist aber

$$\begin{aligned} \alpha\chi'(\alpha) &= k\alpha^k - (k-1)c_{k-1}\alpha^{k-1} - \dots - 2c_2\alpha^2 - c_1\alpha \text{ und} \\ (n-k)\chi(\alpha) &= (n-k)\alpha^k - (n-k)c_{k-1}\alpha^{k-1} - \dots - (n-k)c_1\alpha - (n-k)c_0 \end{aligned}$$

und damit

$$\psi(\alpha) = n\alpha^k - (n-1)c_{k-1}\alpha^{k-1} - \dots - (n-k+1)c_1\alpha - (n-k)c_0.$$

Nach Multiplikation mit  $\alpha^{n-k}$  erhalten wir

$$n\alpha^n = (n-1)c_{k-1}\alpha^{n-1} - \dots - (n-k+1)c_1\alpha^{n-k+1} - (n-k)c_0\alpha^{n-k}$$

also ist auch  $(n\alpha^n)_{n \geq 0}$  eine Lösung.

Falls  $\alpha$  eine dreifache Nullstelle von  $\chi(z)$  ist, ist  $\alpha$  eine Doppelnullestelle von  $\psi(z)$  und wir wenden die Abbildung  $\chi \mapsto \psi$  ein zweites Mal, diesmal auf  $\psi$ , an. Induktiv folgt damit: falls  $\alpha$  eine  $m$ -fache Nullstelle von  $\chi(z)$  ist, dann sind  $(\alpha^n)_{n \geq 0}, (n\alpha^n)_{n \geq 0}, \dots, (n^m \alpha^n)_{n \geq 0}$  Lösungen.

Diese  $k$  Lösungen sind linear unabhängig, also handelt es sich wegen Lemma 4.1 um eine Basis des Lösungsraums.  $\square$

Die allgemeine Lösung von (4.1) hat also die Form

$$x_n = \sum_{i=1}^r \sum_{j=0}^{m_i-1} c_{i,j} n^j \alpha_i^n$$

wobei die  $c_{i,j}$  Konstanten sind, die von den Anfangswerten abhängen.

*Beispiel 4.3.* Mit Hilfe des obigen Satzes lässt sich leicht eine explizite Darstellung der Fibonacci-Zahlen berechnen. Das charakteristische Polynom von  $F_n = F_{n-1} + F_{n-2}$  ist  $\chi(z) = z^2 - z - 1$ . Dieses hat die Nullstellen

$$z_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1 \pm \sqrt{5}}{2},$$

woraus sich die allgemeine Lösung

$$F_n = c_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

ergibt, wobei  $c_1$  und  $c_2$  von den Anfangsbedingungen abhängen. Für  $F_0 = 0$  und  $F_1 = 1$  ergibt sich das lineare Gleichungssystem

$$\begin{aligned} c_1 + c_2 &= 0 \\ c_1 \left( \frac{1 + \sqrt{5}}{2} \right) + c_2 \left( \frac{1 - \sqrt{5}}{2} \right) &= 1 \end{aligned}$$

für  $c_1$  und  $c_2$ , das die Lösung  $c_1 = \frac{1}{\sqrt{5}}$ ,  $c_2 = -\frac{1}{\sqrt{5}}$  hat. Wir erhalten also die Darstellung

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

*Bemerkung 4.1.* Die obige Darstellung der Fibonacci-Zahlen ist nicht nur von rein theoretischem Interesse, sie bietet auch einen effizienteren Algorithmus zur Berechnung von  $F_n$ . Der direkt auf der Definition  $F_n = F_{n-1} + F_{n-2}$  basierende Algorithmus benötigt  $\Theta(n)$  Zeit zur Berechnung von  $F_n$ . Zur Verwendung der obigen Darstellung muss im Wesentlichen  $a^n$  (für  $a = \frac{1 \pm \sqrt{5}}{2}$ ) berechnet werden. Das ist in Zeit  $O(\log n)$  wie folgt möglich: zunächst berechnen wir  $a^n$  für hinreichend viele Zweierpotenzen  $n$ , also  $a^0, a^1, a^2, a^4, a^8, \dots$ . Aus  $a^i$  kann  $a^{2i} = (a^i)^2$  mit einer einzigen Multiplikation berechnet werden. Wir schreiben dann  $n$  binär und multiplizieren die entsprechenden Potenzen, also z.B.  $(37)_b = 100101$  und damit  $a^{37} = a^{32} a^4 a^1$ .

Zum Abschluss dieses Abschnitts über lineare Rekursionsgleichungen mit konstanten Koeffizienten wollen wir nun auch noch den inhomogenen Fall betrachten. Dieser ist leicht zu behandeln.

**Definition 4.5.** Eine *inhomogene lineare Rekursionsgleichung mit konstanten Koeffizienten  $k$ -ter Ordnung* ist von der Form

$$x_n = c_{k-1}x_{n-1} + \dots + c_0x_{n-k} + d \text{ für } n \geq k \quad (4.2)$$

**Satz 4.3.** Sei  $K$  ein Körper, seien  $c_0, \dots, c_{k-1}, d \in K$ , dann hat (4.2) die Lösung  $(x_n)_{n \geq 0}$  definiert durch  $x_n = y_n - \frac{d}{C-1}$  wobei  $C = \sum_{i=0}^{k-1} c_i$  und  $y_n$  ist Lösung von

$$\begin{aligned} y_n &= x_n + \frac{d}{C-1} \text{ für } 0 \leq n < k \\ y_n &= c_{k-1}y_{n-1} + \dots + c_0y_{n-k} \text{ für } n \geq k \end{aligned}$$

*Beweis.* Wir gehen mit Induktion nach  $n$  vor. Der Fall  $n < k$  folgt direkt aus der Definition von  $x_n$  und  $y_n$ . Für  $n \geq k$  haben wir

$$\begin{aligned} x_n &= c_{k-1}x_{n-1} + \dots + c_0x_{n-k} + d = c_{k-1}\left(y_{n-1} - \frac{d}{C-1}\right) + \dots + c_0\left(y_{n-k} - \frac{d}{C-1}\right) + d \\ &= c_{k-1}y_{n-1} + \dots + c_0y_{n-k} - C \cdot \frac{d}{C-1} + d = y_n - \frac{d}{C-1}. \end{aligned}$$

□

### 4.3 Die Substitutionsmethode

Als Substitutionsmethode bezeichnet man die aus den folgenden beiden Schritten bestehende Vorgehensweise zum Auflösen einer Rekursionsgleichung:

1. Errate die Form der Lösung.
2. Beweise dass die Form korrekt ist.

Üblicherweise wird der zweite Schritt mit Induktion gemacht wobei die Verwendung der Induktionshypothese die Form einer *Substitution* eines  $T$ -Terms durch die erratene rechte Seite hat. Man verwendet dabei einen Lösungsansatz mit unbekanntem Konstanten und bestimmt diese Konstanten erst im Lauf des zweiten Schritts (bzw. deren Existenz zu zeigen). Eine ähnliche Vorgehensweise (Beweisansatz mit unbekanntem Konstanten, die erst nach Abschluss des Beweises bestimmt werden) ist in vielen Situationen zum Beweis von Abschätzungen nützlich.

*Beispiel 4.4.* Eine etwas vereinfachte Variante der Rekursionsgleichung von Sortieren durch Verschmelzen ist

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n.$$

Wir wollen uns jetzt nicht mehr für die exakte Lösung interessieren, sondern nur noch für eine asymptotische Lösung. Deswegen sind die konkreten Zahlenwerte eines endlichen Anfangs von  $T$  irrelevant und werden gar nicht mehr angegeben. Zunächst erraten wir die obere Schranke  $T(n) = O(n \log n)$ . Wir müssen also zeigen dass  $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn \log n$ . Die Konstanten  $c$  und  $n_0$  lassen wir noch offen und setzen zu einem Induktionsbeweis an.

Bezüglich des Induktionsanfangs bemerken wir, dass die Funktion  $n \mapsto n \log n$  die Nullstellen 0 und 1 hat, danach ist sie positiv. Nachdem wir nicht garantieren können dass  $T(0) = T(1) = 0$  müssen wir uns auf  $n_0 \geq 2$  einschränken. Dann muss als Induktionsbasis ein hinreichend großes Intervall  $[n_0, b[$  gewählt werden, so dass sowohl  $n \mapsto \lceil \frac{n}{2} \rceil$  als auch  $n \mapsto \lfloor \frac{n}{2} \rfloor$ , angewandt auf  $n \geq b$  immer noch größer gleich  $n_0$  ist. Offensichtlich reicht dafür  $b = 2n_0$ . Damit schränken wir uns ein auf  $c$  mit  $T(m) \leq cm \log m$  für  $m \in [n_0, 2n_0[$ .

Für den Induktionsschritt sei  $n \geq 2n_0$ . Wir nehmen an dass  $T(m) \leq cm \log m$  für alle  $m \in [n_0, n[$  und setzen an mit

$$T(n) \leq c \left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + n = (*).$$

Falls  $n = 2k$ , dann

$$\begin{aligned} (*) &= cn \log \frac{n}{2} + n = cn \log n - cn \log 2 + n. \\ &= cn \log n - cn + n \end{aligned}$$

Nun schränken wir uns ein auf  $c \geq 1$  und erhalten damit

$$\leq cn \log n.$$

Falls  $n = 2k + 1$ , dann

$$(*) = c(k+1) \log(k+1) + ck \log k + n \leq cn \log(k+1) + n$$

und da  $k = \frac{n-1}{2}$  ist  $k+1 = \frac{n+1}{2}$  und damit

$$= cn \log \frac{n+1}{2} + n = cn \log(n+1) - cn + n.$$

Wir wollen  $(*) \leq cn \log n$  erreichen und müssen uns dazu wieder einschränken auf  $c \geq 1$ . Das allein reicht aber noch nicht, der Term  $-cn$  muss zusätzlich zu  $n$  auch  $cn(\log(n+1) - \log n)$  entfernen. Wir brauchen also ein  $n_0$  so dass für alle  $n \geq n_0$  gilt

$$cn \log(n+1) - cn + n \leq cn \log n, \text{ d.h.}$$

$$\log(n+1) - 1 + \frac{1}{c} \leq \log n, \text{ d.h.}$$

$$\log(n+1) - \log n \leq 1 - \frac{1}{c}, \text{ d.h.}$$

$$\log \frac{n+1}{n} \leq 1 - \frac{1}{c}.$$

Wir müssen also  $c, n_0$  finden so dass die folgenden Bedingungen erfüllt sind:

1.  $n_0 \geq 2$
2.  $\forall m \in [n_0, 2n_0[ : T(m) \leq cm \log m$
3.  $c \geq 1$
4.  $\forall n \geq 2n_0 : \log \frac{n+1}{n} \leq 1 - \frac{1}{c}$

Dazu wählen wir zuerst ein  $c > 1$ , dann gibt es  $n_0$  das 4. wahr macht da  $c > 1$  impliziert dass  $1 - \frac{1}{c} > 0$  und  $\frac{n+1}{n} \searrow 1$  und damit  $\log \frac{n+1}{n} \searrow 0$  für  $n \rightarrow \infty$ . Falls dieses  $n_0$  zu klein war erhöhen wir es um  $n_0 \geq 2$  zu erfüllen wodurch 4. beibehalten wird. Es bleibt noch, 2. zu erfüllen. Das geschieht durch hinreichende Erhöhung von  $c$  bei gleichbleibendem  $n_0$  und die Beobachtung dass dadurch 1., 3. und 4. beibehalten werden.

Damit haben wir also gezeigt dass  $c$  und  $n_0$  existieren die den Bedingungen 1.-4. genügen und damit dass  $T(n) = O(n \log n)$ .

*Beispiel 4.5.* Sei

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1.$$

Eine Vermutung ist  $T(n) = O(n)$ , d.h.  $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn$ . Auf direkte Weise würde das zum folgenden Ansatz für den Induktionsschritt führen:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1$$

Wir wollen  $T(n) \leq cn$  erreichen was offensichtlich unmöglich ist. Jetzt ist aber  $T(n) = O(n)$  trotzdem wahr was durch den folgenden, verbesserten, Ansatz auch gezeigt werden kann:  $\exists c > 0, d > 0, n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn - d$ . Dann erhalten wir

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d + c \left\lceil \frac{n}{2} \right\rceil - d + 1 = cn - 2d + 1 \leq cn - d$$

unter der Voraussetzung  $d \geq 1$ . Der Induktionsanfang wird wie oben behandelt und so erhält man  $T(n) = O(cn - d) = O(n)$ .

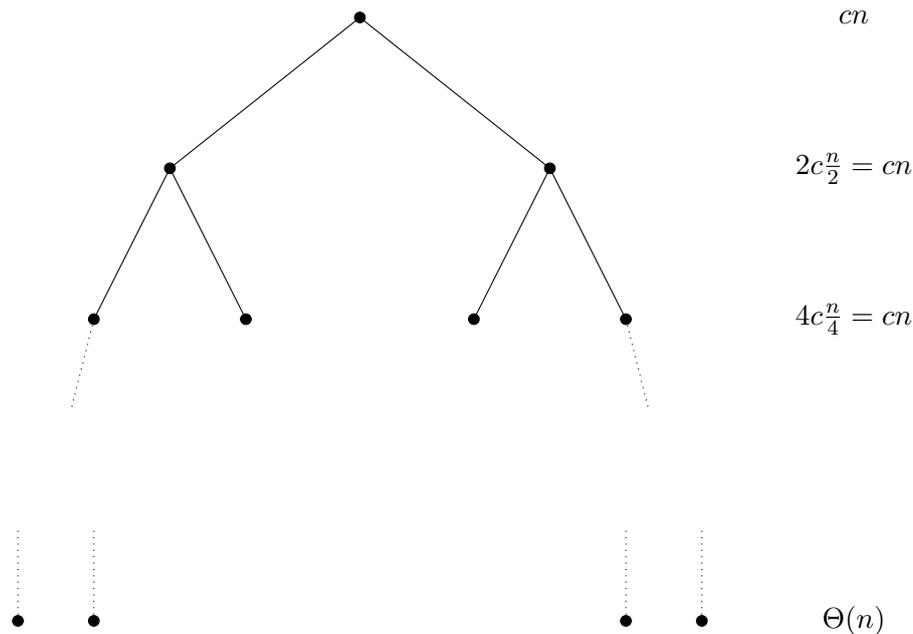


Abbildung 4.1: Rekursionsbaum von  $T(n) = 2T(\frac{n}{2}) + cn$

## 4.4 Die Rekursionsbaummethode

Die Rekursionsbaummethode besteht im Aufzeichnen des durch eine Rekursionsgleichung induzierten Rekursionsbaums sowie in der anschließenden Summierung der im gesamten Baum anfallenden Kosten. Sie wird oft verwendet, um zu einer Vermutung zu gelangen, die danach mit der Substitutionsmethode bewiesen wird. Deshalb wird mit Rekursionsbäumen oft recht ungenau gearbeitet, z.B. lässt man die Rundungsoperatoren  $\lfloor \cdot \rfloor$  und  $\lceil \cdot \rceil$  weg oder man ersetzt Ausdrücke wie  $\Theta(f(n))$  durch  $cf(n)$ .

*Beispiel 4.6.* Die Rekursion von Sortieren durch Verschmelzen war

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}$$

Wir vereinfachen diese zu  $T(n) = 2T(\frac{n}{2}) + cn$  und zeichnen den Rekursionsbaum auf, siehe Abbildung 4.1.

In der ersten Ebene, d.h. am Wurzelknoten, treten Kosten von  $cn$  auf. In der zweiten Ebene treten an jedem Knoten Kosten von  $c\frac{n}{2}$  auf. Da es in der zweiten Ebene zwei Knoten gibt, treten insgesamt in der zweiten Ebene Kosten von  $cn$  auf. In der dritten Ebene gibt es vier Knoten mit Kosten jeweils  $c\frac{n}{4}$ , also hat auch diese Ebene Kosten von  $cn$ , usw. Jede Ebene hat also Kosten  $cn$ . Wie viele Ebenen gibt es? Bis  $n$  auf eine Konstante (z.B. konkret auf 1) reduziert wurde benötigen wir  $\log n$  Ebenen. Der Rekursionsbaum ist also ein vollständiger Binärbaum der Tiefe  $\log n$  wobei wir auch hier, im Sinne einer vereinfachenden Annahme, die Tatsache vernachlässigen dass  $\log n$  nicht notwendigerweise eine natürliche Zahl ist und nicht klar ist was mit einem Baum reeller Tiefe gemeint sein soll. Ein solcher Baum hat  $\Theta(n)$  Blätter. So kommen wir also zur Vermutung  $T(n) = cn \log n + \Theta(n) = \Theta(n \log n)$ .

*Beispiel 4.7.* Betrachten wir nun die vereinfachte Rekursionsgleichung

$$T(n) = 3T(\frac{n}{4}) + cn^2$$

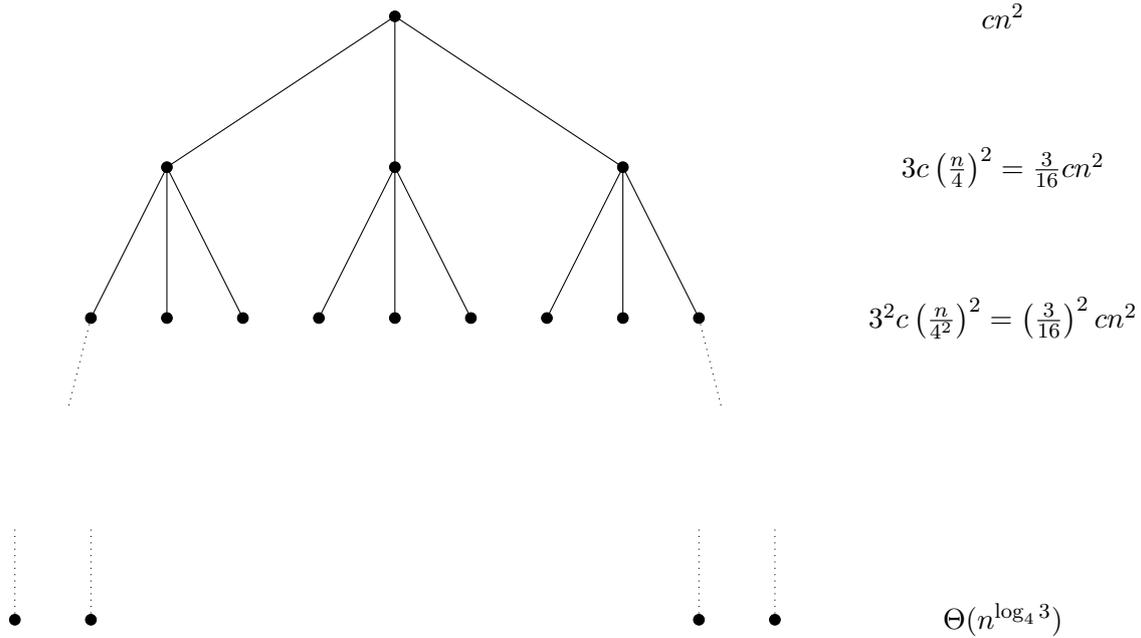


Abbildung 4.2: Rekursionsbaum von  $T(n) = 3T(\frac{n}{4}) + cn^2$

Der Rekursionsbaum für diese Gleichung ist in [Abbildung 4.2](#) zu finden. Die erste Ebene hat Kosten  $cn^2$ , die zweite  $\frac{3}{16}cn^2$ , die dritte  $(\frac{3}{16})^2 cn^2$  usw. Dieser Rekursionsbaum ist ein vollständiger Baum der Arität 3 mit Tiefe  $\log_4 n$ , er hat also  $3^{\log_4 n} = n^{\log_4 3}$  Blätter. Nun beobachten wir dass

$$\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} < cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + n^{\log_4 3} = cn^2 \frac{1}{1 - \frac{3}{16}} + n^{\log_4 3} = O(n^2)$$

und erhalten somit die Vermutung  $T(n) = O(n^2)$ .

## 4.5 Die Mastermethode

Summenbildungen über Rekursionsbäume, wie sie in den beiden Beispielen im vorherigen Abschnitt auftreten, folgen immer dem selben Muster. Wir werden nun einen allgemeinen Satz über solche Summen beweisen, der dann erlaubt die Lösungen der allermeisten Teile-und-herrsche Rekursionsgleichungen als Korollare zu erhalten. Dazu betrachten wir  $a_0, a_1 \in \mathbb{N}$  mit  $a = a_0 + a_1 \geq 1$ ,  $b > 1$  und eine Rekursionsgleichung der Form

$$T(n) = a_0 T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + a_1 T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) \tag{4.3}$$

Da wir uns nur für asymptotische Lösungen interessieren, entfällt die Angabe von Anfangswerten. Der Anfang des Rekursionsbaums der Gleichung (4.3) ist in [Abbildung 4.3](#) dargestellt. Um das iterierte auf- und abrunden darzustellen verwenden wir Zeichenketten im Alphabet  $\{0, 1\}$  wobei 0 für Abrunden und 1 für Aufrunden steht.

**Definition 4.6.**  $\{0, 1\}^*$  ist die Menge aller endlich langen aus 0 und 1 bestehenden Zeichenketten (Wörtern) inklusive dem Leerwort  $\varepsilon$ . Die Länge  $|w|$  eines Wortes  $w \in \{0, 1\}^*$  ist die

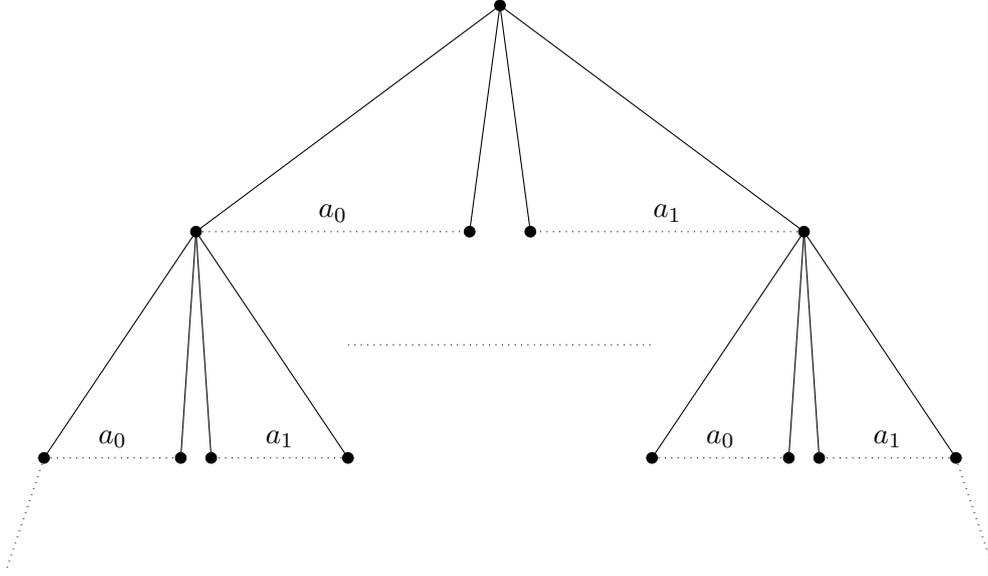


Abbildung 4.3: Anfang des Rekursionsbaums von Gleichung (4.3)

Anzahl der Zeichen aus denen  $w$  besteht. Die Länge des Leerwortes ist  $|\varepsilon| = 0$ . Für  $w \in \{0, 1\}^*$  bezeichnet  $n_0(w)$  die Anzahl von Nullern in  $w$  und  $n_1(w)$  die Anzahl von Einsen in  $w$ . Für einen Buchstaben  $x$  und ein  $i \in \mathbb{N}$  bezeichnet  $x^i$  das Wort das aus  $i$  Vorkommen des Buchstaben  $x$  besteht.

**Definition 4.7.** Sei  $b > 1$ . Wir definieren für  $n \in \mathbb{N}$  und  $w \in \{0, 1\}^*$  die Zahl  $n_w \in \mathbb{N}$  als

$$n_w = \begin{cases} n & \text{falls } w = \varepsilon \\ \lfloor \frac{n_w}{b} \rfloor & \text{falls } w = 0v \\ \lceil \frac{n_w}{b} \rceil & \text{falls } w = 1v \end{cases}$$

Jeder Knoten im Baum entspricht also  $T$  angewandt auf ein bestimmtes  $n_w$ . Ein  $w$  kommt im Baum an  $a_0^{n_0(w)} a_1^{n_1(w)}$  vielen Stellen vor. Die lokalen Kosten an einem Knoten der  $T(n_w)$  entspricht sind  $f(n_w)$ . Wir wollen diese lokalen Kosten über alle Knoten im Baum summieren.

Wie tief ist dieser Baum? Falls  $n$  eine Potenz von  $b$  ist, und somit niemals gerundet wird, ist klar: nach  $\log_b n$  Schritten in die Tiefe ist  $T(1)$  erreicht. Aber auch der Fall wo  $n$  keine Potenz von  $b$  ist, ist nicht wesentlich komplizierter:  $n_w$  verhält sich bis auf eine additive Konstante so wie  $\frac{n}{b^{|w|}}$ , siehe Abbildung 4.4.

**Lemma 4.2.** Sei  $b > 1$ , dann existiert ein  $c > 0$  so dass für alle  $n \in \mathbb{N}$  und  $w \in \{0, 1\}^*$ :  $\frac{n}{b^{|w|}} - c < n_w < \frac{n}{b^{|w|}} + c$ .

*Beweis.* Sei  $|w| = j$ , dann ist  $n_{0^j} \leq n_w \leq n_{1^j}$ . Wir behaupten weiters dass

$$n_{0^j} \geq \frac{n}{b^j} - \sum_{i=0}^{j-1} \frac{1}{b^i} \quad \text{und} \quad n_{1^j} \leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i}.$$

Wir zeigen die Ungleichung für  $n_{1^j}$  mit Induktion nach  $j$ . Falls  $j = 0$ , dann ist  $n_\varepsilon = n$ . Für den Induktionsschritt gilt

$$n_{1^{j+1}} = \lceil \frac{n_{1^j}}{b} \rceil \leq \frac{n_{1^j}}{b} + 1 \stackrel{\text{IH}}{\leq} \frac{1}{b} \left( \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \right) + 1 = \frac{n}{b^{j+1}} + \sum_{i=0}^j \frac{1}{b^i}.$$

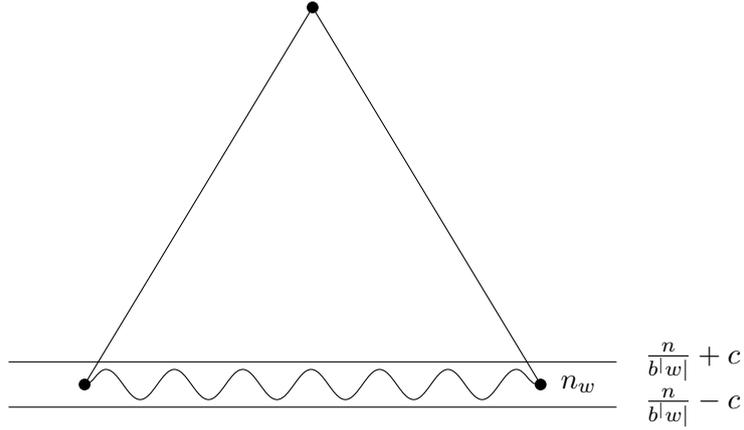


Abbildung 4.4: Illustration von Lemma 4.2 für  $w \in \{0, 1\}^j$

Analog dazu wird die Ungleichung für  $n_{0^j}$  gezeigt. Das Resultat folgt nun aus

$$\sum_{i=0}^{j-1} \frac{1}{b^i} < \sum_{i=0}^{\infty} \left(\frac{1}{b}\right)^i = \frac{1}{1 - \frac{1}{b}} = \frac{b}{b-1} = c.$$

□

Für die Tiefe  $d(n)$  des Baums können wir nun eine beliebige Funktion wählen, die  $\frac{n}{b^{d(n)}} = \Theta(1)$  erfüllt. Dann ist nämlich nach Lemma 4.2 auch für alle  $w \in \{0, 1\}^{d(n)}$  erreicht dass  $n_w = \Theta(1)$  und damit auch für alle  $w \in \{0, 1\}^{d(n)}$  dass  $T(n_w) = \Theta(1)$ . Wir setzen  $d(n) = \lfloor \log_b n \rfloor$  und beobachten dass  $\frac{n}{b^{d(n)}} = \Theta(1)$ . Dementsprechend definieren wir die Indizes der inneren Knoten und die Indizes der Blattknoten als:

**Definition 4.8.**  $X_I(n) = \{0, 1\}^{\leq d(n)-1}$  und  $X_B(n) = \{0, 1\}^{d(n)}$ .

Die Kosten des gesamten Baums setzen sich zusammen aus den Kosten an den inneren Knoten  $\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$  und den Kosten an den Blättern. Jedes Blatt verursacht konstante Kosten. Für die Bestimmung der Anzahl der Blätter spielt der Unterschied zwischen  $a_0$  und  $a_1$  keine Rolle und somit können wir zu diesem Zweck den Baum als vollständigen Baum der Arität  $a$  mit Tiefe  $d(n)$  betrachten. Dieser hat  $a^{d(n)} = a^{\lfloor \log_b n \rfloor} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$  Blätter. Also erhalten wir

$$T(n) = \Theta(n^{\log_b a}) + \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w).$$

für die Summe über den gesamten Baum.

Eine weitere nützliche Beobachtung ist: Falls wir eine Funktion  $\varphi$  schichtweise summieren wollen die nur von der Länge von  $w$ , nicht aber von  $w$  selbst abhängt, dann spielt der Unterschied zwischen  $a_0$  und  $a_1$  ebenfalls keine Rolle und wir können ebenfalls den Baum als einen vollständigen Baum mit Arität  $a$  betrachten und entsprechend aufsummieren, d.h.

$$\sum_{w \in \{0, 1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = a^j \varphi(j).$$

Damit ist also

$$\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = \sum_{j=0}^{d(n)-1} a^j \varphi(j).$$

**Satz 4.4** (Master-Theorem für Teile-und-herrsche-Rekursionsgleichungen). *Seien  $a_0, a_1 \in \mathbb{N}$  mit  $a = a_0 + a_1 \geq 1$ , sei  $b > 1$  und sei  $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ . Dann hat die Rekursionsgleichung*

$$T(n) = a_0 T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + a_1 T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$$

die folgende asymptotische Lösung.

1. Falls  $f(n) = O(n^{\log_b a - \varepsilon})$  für ein  $\varepsilon > 0$ , dann  $T(n) = \Theta(n^{\log_b a})$ .
2. Falls  $f(n) = \Theta(n^{\log_b a})$ , dann  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. Falls es ein  $c < 1$  gibt, so dass für alle bis auf endlich viele  $n \in \mathbb{N}$  die Ungleichung  $a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil) \leq c f(n)$  gilt, dann ist  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  für ein  $\varepsilon > 0$  und  $T(n) = \Theta(f(n))$ .

Wie man sehen kann ist für dieses Resultat vor allem  $a$  wesentlich,  $a_0$  und  $a_1$  spielen eine untergeordnete Rolle. Deshalb werden solche Rekursionsgleichungen oft auch geschrieben als  $T(n) = aT(\frac{n}{b}) + f(n)$  wobei jedes der  $a$  Vorkommen von  $\frac{n}{b}$  für  $\lceil \frac{n}{b} \rceil$  oder  $\lfloor \frac{n}{b} \rfloor$  steht.

*Beweis.* Sei  $g(n) = \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$ . In Fall 1 haben wir

$$g(n) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a - \varepsilon}\right) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}}\right)^{\log_b a - \varepsilon}\right)$$

weil  $n_w = \Theta(\frac{n}{b^{|w|}})$  und weiters

$$\begin{aligned} &= O\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon}\right) = O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a - \varepsilon}}\right)^j\right) \\ &= O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} (b^\varepsilon)^j\right) = O\left(n^{\log_b a - \varepsilon} \frac{b^{\varepsilon d(n)} - 1}{b^\varepsilon - 1}\right) \\ &= O(n^{\log_b a - \varepsilon} n^\varepsilon) = O(n^{\log_b a}) \end{aligned}$$

weil  $b^{d(n)} = \Theta(n)$ . Insgesamt erhalten wir also  $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$ .

In Fall 2 haben wir

$$g(n) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a}\right) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}}\right)^{\log_b a}\right)$$

weil  $n_w = \Theta(\frac{n}{b^{|w|}})$  und weiters

$$\begin{aligned} &= \Theta\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) = \Theta\left(n^{\log_b a} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a}}\right)^j\right) \\ &= \Theta\left(n^{\log_b a} d(n)\right) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \log n). \end{aligned}$$

Insgesamt erhalten wir also  $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$ .

In Fall 3 ist  $c f(n) \geq a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil)$  für alle  $n \geq n_0$ . Sei nun  $d_0(n) = d(n) - p$  wobei  $p \in \mathbb{N}$  so gewählt ist, dass  $\frac{n}{b^{d_0(n)}} \geq n_0$  für alle  $n \geq n_0$ . Außerdem ist  $\frac{n}{b^{d_0(n)}} = \Theta(1)$ . Sei  $n \geq n_0$ . Wir zeigen zunächst

$$c^j f(n) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \quad \text{für } j = 0, \dots, d_0(n) \quad (*)$$

mit Induktion nach  $j$ . Falls  $j = 0$ , dann ist  $f(n) = f(n_\varepsilon)$ . Für den Induktionsschritt haben wir

$$\begin{aligned} c^{j+1} f(n) &\geq c \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} (a_0 f(n_{0w}) + a_1 f(n_{1w})) \\ &= \sum_{v \in \{0,1\}^{j+1}} a_0^{n_0(v)} a_1^{n_1(v)} f(n_v). \end{aligned}$$

Dann ist  $c^{d_0(n)} f(n) \geq \sum_{w \in \{0,1\}^{d_0(n)}} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$ , da aber  $\frac{n}{b^{d_0(n)}} = \Theta(1)$  ist wegen Lemma 4.2 auch  $\min\{f(n_w) \mid w \in \{0,1\}^{d_0(n)}\}$  eine Konstante  $q$  und somit  $c^{d_0(n)} f(n) \geq q a^{d_0(n)}$ . Damit erhalten wir

$$f(n) \geq q \left(\frac{a}{c}\right)^{\lfloor \log_b n \rfloor - p} = \Theta\left(\left(\frac{a}{c}\right)^{\log_b n}\right) = \Theta(n^{\log_b \frac{a}{c}}).$$

Nun ist  $n^{\log_b \frac{a}{c}} = n^{\log_b a - \log_b c}$  und da  $c < 1$  war ist  $\log_b c < 0$ , also ist  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  für  $\varepsilon = -\log_b c > 0$ .

Es bleibt zu zeigen dass  $g(n) = \Theta(f(n))$ . Zunächst einmal ist klar dass  $g(n) = \Omega(f(n))$  da  $f(n) = f(n_\varepsilon)$  ja ein Summand von  $g(n)$  ist. Für die andere Richtung betrachten wir den Rekursionsbaum bis zur Tiefe  $d_0(n)$ . Die Anzahl von Blättern ist  $a^{d_0(n)} = a^{\lfloor \log_b n \rfloor - p} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$ . Für  $j \in \{d_0(n), \dots, d(n) - 1\}$  haben wir:  $\forall w \in \{0,1\}^j: \frac{n}{b^{|w|}} = \Theta(1)$ , also  $\forall w \in \{0,1\}^j: n_w = \Theta(1)$ , also  $\forall w \in \{0,1\}^j: f(n_w) = \Theta(1)$ . Weiters ist die Anzahl der  $f(n_w)$ -Summanden an einem Blatt der Tiefe  $d_0(n)$  konstant. Also sind die Kosten an einem Blatt der Tiefe  $d_0(n)$  konstant und wir erhalten:

$$\begin{aligned} g(n) &= \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \leq \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} c^j f(n) \\ &< \Theta(n^{\log_b a}) + f(n) \sum_{j=0}^{\infty} c^j = \Theta(n^{\log_b a}) + \frac{1}{1-c} f(n) = \Theta(f(n)) \end{aligned}$$

da  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  für ein  $\varepsilon > 0$ . Also ist  $g(n) = O(f(n))$ . Insgesamt erhalten wir also  $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$ .  $\square$

**Korollar 4.1.** *Die Rekursionsgleichung*

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

die sowohl die Laufzeit von Sortieren durch Verschmelzen als auch jene von unserem Teile-und-herrsche Algorithmus für das dichteste Punktepaar beschreibt hat die Lösung  $T(n) = \Theta(n \log n)$ .

*Beweis.* In der Notation des master-Theorems gilt  $a = b = 2$  und es trifft der zweite Fall zu. Die Lösung ist also  $\Theta(n^{\log_b a} \log n) = \Theta(n \log n)$ .  $\square$

**Korollar 4.2.** *Die Rekursionsgleichung*

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

die die Laufzeit des einfachen rekursiven Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung  $T(n) = \Theta(n^3)$ .

*Beweis.* In der Notation des master-Theorems ist  $a = 8, b = 2$  und es trifft der erste Fall zu da  $n^2 = O(n^{3-\varepsilon})$  für ein  $\varepsilon > 0$ . Die Lösung ist also  $\Theta(n^{\log_b a}) = \Theta(n^3)$ .  $\square$

**Korollar 4.3.** *Die Rekursionsgleichung*

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

die die Laufzeit des Strassen-Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung  $T(n) = \Theta(n^{2,807\dots})$ .

*Beweis.* In der Notation des master-Theorems ist  $a = 7, b = 2$ . Es ist  $\log_2 7 = 2,807\dots$ . Da  $n^2 = O(n^{2,807\dots-\varepsilon})$  für ein  $\varepsilon > 0$  trifft der erste Fall zu. Die Lösung ist also  $\Theta(n^{\log_b a}) = \Theta(n^{2,807\dots})$ .  $\square$

*Beispiel 4.8.* Auf die Rekursionsgleichung

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \log n$$

ist das master-Theorem in dieser Form nicht anwendbar. Es ist nämlich  $a = b = 2$ , so dass  $n \log n$  mit  $n$  verglichen werden muss. Nun ist aber  $n \log n \neq \Theta(n)$ ,  $n \log n \neq \Omega(n^{1+\varepsilon})$  sowie  $n \log n \neq O(n^{1-\varepsilon})$ . Damit trifft keiner der Fälle des master-Theorems zu. Diese Rekursionsgleichung hat die asymptotische Lösung  $T(n) = \Theta(n \log^2 n)$ .



# Kapitel 5

## Datenstrukturen

In diesem Kapitel werden wir uns mit Datenstrukturen beschäftigen. Eine Datenstruktur ist eine Struktur zur Speicherung und Organisation von Daten, die typischerweise gewisse Operationen (auf effiziente Weise) zur Verfügung stellt. Eine einfache Datenstruktur, das Datenfeld, haben wir bereits kennengelernt. Es erlaubt Lese- und Schreibzugriff auf ein beliebiges Element dessen Index bekannt ist in konstanter Zeit. Algorithmen und Datenstrukturen gehen Hand in Hand: jede Datenstruktur benötigt Algorithmen, welche die gewünschten Operationen zur Verfügung stellen und umgekehrt: oft sind gewisse Datenstrukturen notwendig, damit ein Algorithmus eine bestimmte Laufzeit erreicht. So haben wir zum Beispiel im teile-und-herrsche Algorithmus zur Lösung des dichtesten Punktepaarproblems (Algorithmus 10) eine Menge auf eine Weise repräsentiert, die das Durchlaufen anhand zweier unterschiedlicher Ordnungen erlaubt hat. Auch diese Repräsentation kann als Datenstruktur betrachtet werden.

Datenstrukturen sind darüber hinaus auch deswegen von großer Bedeutung in der Informatik, weil die Betrachtung von Berechnungsproblemen und Algorithmen als Lösungen dieser zwar für viele aber bei weitem nicht für alle Anwendungen adäquat ist. Oft befindet man sich in der Praxis in einer Situation die dynamisch ist, also nicht durch eine Eingabe-Ausgabe-Relation vollständig beschrieben werden kann.

### 5.1 Das Wörterbuchproblem

In diesem Abschnitt wollen wir das *Wörterbuchproblem* betrachten. Das *Wörterbuchproblem* besteht darin eine möglichst effiziente Organisation einer endlichen Menge von Datensätzen zu finden die abgefragt und verändert(!) werden kann. Wir nehmen dabei an, dass jeder Datensatz  $D$  durch einen eindeutigen Schlüssel  $D.x$  identifiziert wird. Wir wollen, dass unser Wörterbuch  $M$  zumindest die folgenden Operationen unterstützt:

1.  $M.Suche(x)$  gibt jenes  $D$  aus  $M$  zurück dessen Schlüssel  $x$  ist falls es existiert.
2.  $M.Einfügen(D)$  fügt den neuen Datensatz  $D$  zu  $M$  hinzu.
3.  $M.Löschen(x)$  entfernt den Datensatz mit Schlüssel  $x$  aus  $M$ .

Zum Beispiel könnte man an einer Repräsentation aller Studenten dieser Universität interessiert sein. Diese Menge verändert sich im Laufe der Zeit. Die Matrikelnummer kann als eindeutiger Schlüssel dienen. Als Lösung für das Wörterbuchproblem erwarten wir eine geeignete Datenstruktur gemeinsam mit zumindest drei Algorithmen, die die oben beschriebenen Operationen (möglichst effizient) durchführen.

Die einfachste Art ein solches Wörterbuch zu realisieren besteht darin,  $M$  als Datenfeld aufzufassen. Die Laufzeitkomplexität der Operationen (wobei  $n = |M|$ ) ist dann wie folgt:

1.  $M.Suche(x)$  benötigt Zeit  $O(n)$ , d.h. genauer:  $\Theta(n)$  im schlechtesten Fall und  $\Theta(1)$  im besten Fall.
2.  $M.Einfügen(D)$  benötigt Zeit  $\Theta(n)$  da sichergestellt werden muss, dass kein Duplikat erzeugt wird und dafür im schlechtesten Fall ( $D.x$  existiert noch nicht) das gesamte Datenfeld durchlaufen werden muss.
3.  $M.Löschen(x)$  benötigt Zeit  $\Theta(n)$  da  $D$  mit  $D.x = x$  zunächst gefunden werden muss und dann das Datenfeld verkürzt werden muss.

Eine bessere Darstellung von  $M$  besteht darin, ein nach Schlüssel (aufsteigend) sortiertes Datenfeld zu verwenden. Dann haben wir die folgenden Operationen:

1.  $M.Suche(x)$  in Zeit  $O(\log n)$  mit binärer Suche (engl. *binary search*), siehe Algorithmus 11.
2.  $M.Einfügen(D)$  in Zeit  $\Theta(n)$  da das Datenfeld verlängert werden muss.
3.  $M.Löschen(x)$  in Zeit  $\Theta(n)$  da das Datenfeld verkürzt werden muss.

---

#### Algorithmus 11 Binäre Suche

---

**Prozedur** BSUCHE( $A, x$ )

**Antworte** BSUCHEREK( $A, x, 1, A.Länge$ )

**Ende Prozedur**

**Prozedur** BSUCHEREK( $A, x, l, r$ )

**Falls**  $l > r$  **dann**

**Antworte** "nicht gefunden"

**sonst**

$m := \lceil \frac{l+r}{2} \rceil$

**Falls**  $A[m] = x$  **dann**

**Antworte**  $m$

**sonst falls**  $A[m] < x$  **dann**

**Antworte** BSUCHEREK( $A, x, m + 1, r$ )

**sonst**

**Antworte** BSUCHEREK( $A, x, l, m - 1$ )

$\triangleright A[m] > x$

**Ende Falls**

**Ende Falls**

**Ende Prozedur**

---

Mit einem sortierten Datenfeld ist die Suche also effizient, Änderung hingegen nicht.

Eine Datenstruktur in der lokale Änderungen auf effiziente Weise gemacht werden können sind verkettete Listen. Hier unterscheidet man zwischen einfach verketteten und doppelt verketteten Listen. Die Grundidee ist, dass die Elemente einer Liste jeweils einzeln an einer beliebigen Stelle im Speicher abgelegt werden und jedes Element auf das nächste (einfache Verkettung) bzw. auf das nächste und das vorherige verweist (doppelte Verkettung). Diese Verweise auf Speicherpositionen werden als *Zeiger* (engl. *pointer*) bezeichnet. Ein Zeiger verweist entweder auf einen bestimmten Ort im Speicher oder er hat den Wert NIL, den Nullzeiger. Realisiert wird ein

einzelner Knoten durch ein Tupel  $v$  (für Vertex), dessen Elemente im Fall einer einfach verketteten Liste  $v.D$  und  $v.nächster$  sind, im Fall einer doppelt verketteten Liste  $v.D$ ,  $v.nächster$  und  $v.vorheriger$  sind. Der Vorteil einer doppelt verketteten Liste gegenüber einer einfach verketteten besteht darin, dass sie in beide Richtungen durchlaufen werden kann, ihr Nachteil darin, dass sie (konstant) mehr Speicherplatz benötigt. Doppelt verkettete Listen erlauben Einfügen und löschen in konstanter Zeit durch Umsetzen der pointer. Eine doppelt verkettete Liste als Darstellung von  $M$  erlaubt die folgenden Operationen:

1.  $M.Suche(x)$  benötigt Zeit  $O(n)$  genauer:  $\Theta(n)$  im schlechtesten Fall,  $\Theta(1)$  im besten Fall und  $\Theta(n)$  im Durchschnittsfall.
2.  $M.Einfügen(D)$  in Zeit  $\Theta(n)$  da sichergestellt werden muss, dass kein Duplikat erzeugt wird
3.  $M.Löschen(x)$  wie die Suche in Zeit  $O(n)$  da  $D$  mit  $D.x = x$  gefunden werden muss.

Doppelt verkettete Listen erlauben zusätzlich noch die folgenden effizienten Operationen:

- 1'.  $M.Einfügen(D)$  in Zeit  $\Theta(1)$  unter der Annahme dass  $D$  noch nicht in  $M$  vorkommt.
- 2'.  $M.Löschen(v)$  in Zeit  $\Theta(1)$  durch Umsetzen der Zeiger unter der Voraussetzung dass  $v$  ein Element von  $M$  ist.

Bei verketteten Listen ist zwar im Vergleich zum unsortierten Datenfeld nichts gewonnen was die Suche angeht, allerdings erhalten wir bei den dynamischen Operationen Einfügen und Löschen neue Möglichkeiten.

## 5.2 Suchbäume

Eine gute Lösung für das Wörterbuchproblem besteht in der Verwendung von *Suchbäumen*. Ein binärer Baum wird im Speicher, ähnlich einer verketteten Liste, so abgelegt, dass jeder Knoten, zusätzlich zu seinem eigentlichen Inhalt dem Datensatz, noch zwei Zeiger enthält: einen auf das linke Kind und einen auf das rechte Kind. Wir schreiben dafür  $v.D$ ,  $v.links$  und  $v.rechts$ . Für Blätter werden diese beiden Zeiger auf NIL gesetzt.

**Definition 5.1.** Ein Suchbaum ist ein Baum der die folgende *Suchbaumeigenschaft* hat: für alle Knoten  $v$  gilt:

1. Für jeden Knoten  $w$  im linken Teilbaum von  $v$  gilt:  $w.D.x < v.D.x$ .
2. Für jeden Knoten  $w$  im rechten Teilbaum von  $v$  gilt:  $w.D.x > v.D.x$ .

Die Suche in einem Suchbaum funktioniert dann wie in Algorithmus 12 beschrieben. Man beachte die Ähnlichkeit dieses Algorithmus zur binären Suche in einem sortierten Datenfeld. Algorithmus 12 hat Laufzeitkomplexität  $O(h)$  wobei  $h$  die Höhe, d.h. die Länge des längsten Pfades, des Baums ist.

Die Liste aller Elemente die der Suchbaum enthält kann in aufsteigender Reihenfolge ausgegeben werden, indem er in *symmetrischer Reihenfolge* (engl. *inorder*) durchlaufen wird, also zuerst der linke Teilbaum, dann der aktuelle Knoten, dann der rechte Teilbaum, siehe Algorithmus 13.

Ein einfacher Algorithmus zum Einfügen eines Elements  $D$  ist in Algorithmus 14 angegeben.

---

**Algorithmus 12** Suche in einem Suchbaum

---

**Prozedur** SUCHE( $v, x$ )  
  **Falls**  $v = \text{NIL}$  **dann**  
    **Antworte** “nicht gefunden”  
  **sonst falls**  $v.D.x = x$  **dann**  
    **Antworte**  $v.D$   
  **sonst falls**  $v.D.x < x$  **dann**  
    **Antworte** SUCHE( $v.rechts, x$ )  
  **sonst**  $\triangleright v.D.x > x$   
    **Antworte** SUCHE( $v.links, x$ )  
  **Ende Falls**  
**Ende Prozedur**

---

---

**Algorithmus 13** Durchlaufen eines Suchbaums in symmetrischer Reihenfolge

---

**Prozedur** SBAUSGABE( $v$ )  
  **Falls**  $v \neq \text{NIL}$  **dann**  
    SBAUSGABE( $v.links$ )  
    AUSGABE( $v.D$ )  
    SBAUSGABE( $v.rechts$ )  
  **Ende Falls**  
**Ende Prozedur**

---

---

**Algorithmus 14** Einfügen in einen Suchbaum

---

**Prozedur** EINFÜGEN( $v, D$ )  
  **Falls**  $v = \text{NIL}$  **dann**  $\triangleright$  Leerer Suchbaum wird initialisiert  
    Sei  $v_0$  neuer Knoten  
     $v_0.D := D$   
     $v_0.links := \text{NIL}$   
     $v_0.rechts := \text{NIL}$   
    **Antworte**  $v_0$   
  **sonst falls**  $v.D.x = D.x$  **dann**  
    **Antworte** “Schlüssel existiert bereits”  
  **sonst falls**  $v.D.x < D.x$  **dann**  
     $v.rechts := \text{EINFÜGEN}(v.rechts, D)$   
    **Antworte**  $v$   
  **sonst**  $\triangleright v.D.x > D.x$   
     $v.links := \text{EINFÜGEN}(v.links, D)$   
    **Antworte**  $v$   
  **Ende Falls**  
**Ende Prozedur**

---

Die Vorgehensweise zum Löschen eines Elements wird am besten zunächst an einem Beispiel illustriert. Angenommen wir wollen aus dem in Abbildung 5.1 angegebenen Suchbaum das Element mit dem Schlüssel 3 löschen. Dann wird dadurch zunächst einmal die Baumstruktur zerstört und wir erhalten den in Abbildung 5.2 angegebenen Suchbaum. Eine konservative Methode zur Wiederherstellung eines Suchbaums, d.h. eine die möglichst wenig verändert, besteht darin, ein geeignetes Element an die entstandene Lücke zu verschieben. Dafür geeignete Element sind einerseits das Maximum im linken Teilbaum von 3 sowie andererseits das Minimum im rechten

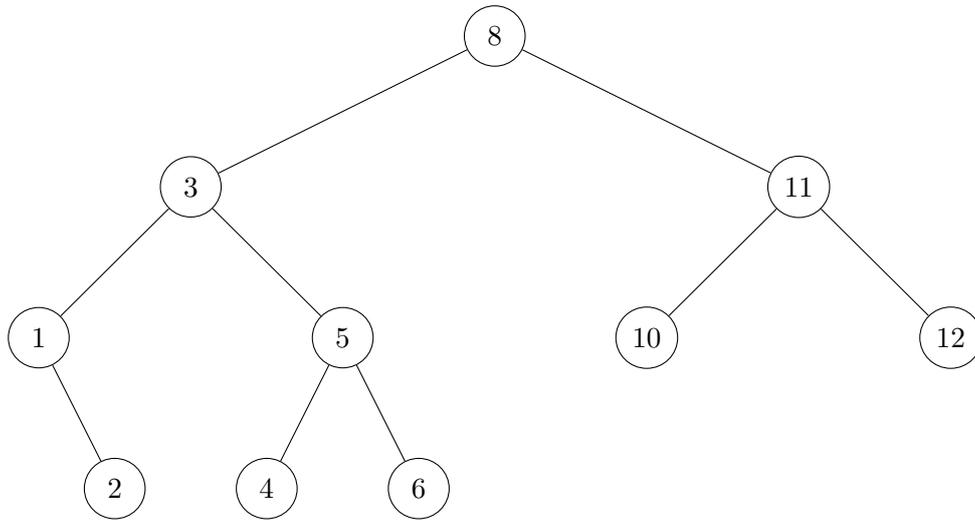


Abbildung 5.1: Suchbaum

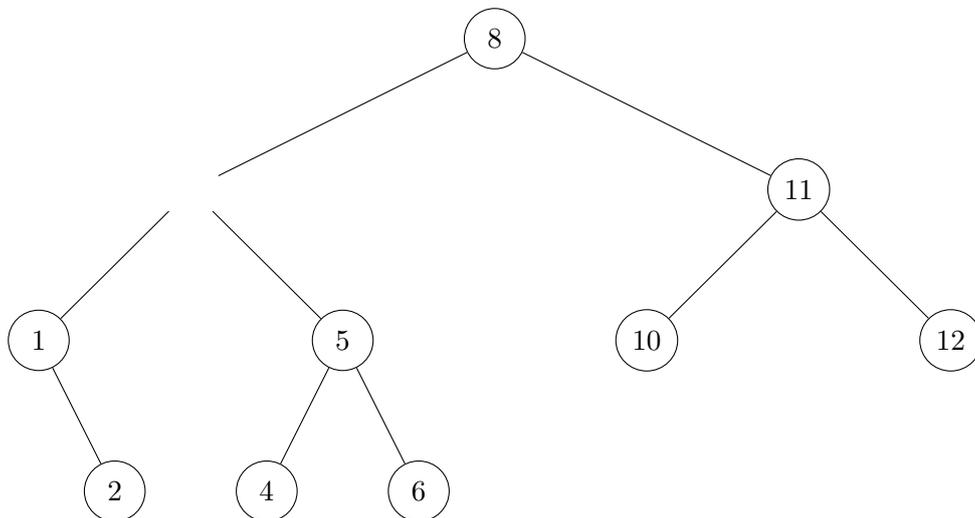


Abbildung 5.2: Suchbaum nach Löschung von 3

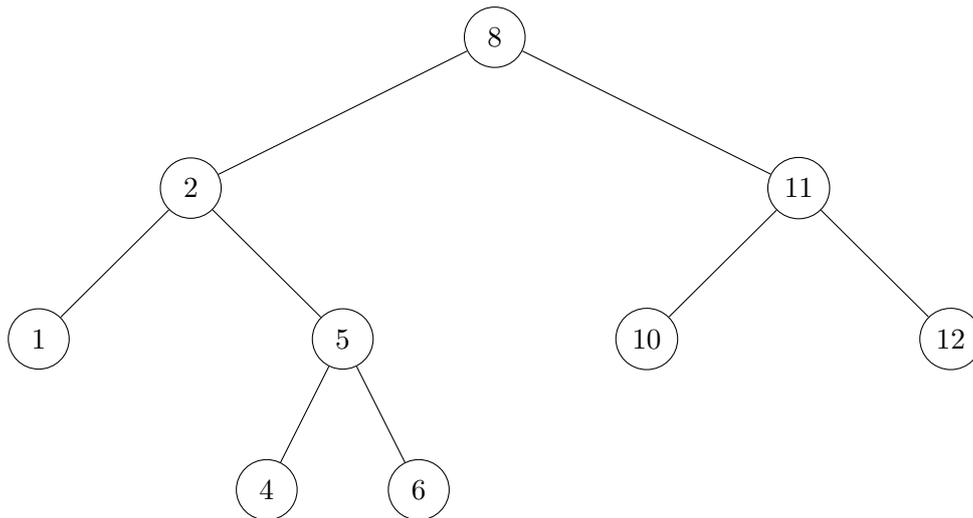


Abbildung 5.3: Suchbaum nach Ersetzung von 3 durch 2

Teilbaum von 3. Wenn wir uns für das Maximum  $m$  des linken Teilbaums entscheiden und an die Stelle des gelöschten Elements setzen sind danach alle Schlüssel im neuen linken Teilbaum kleiner als  $m$  und alle Schlüssel im rechten Teilbaum sind größer als  $m$  da sie ja größer als das gelöschte Element waren und dieses wiederum größer als  $m$ . Das Argument für das Minimum des rechten Teilbaums ist analog. In unserem Beispiel erhalten wir also den in Abbildung 5.3 angegebenen Suchbaum. Als Pseudocode ist sieht dieser Algorithmus wie folgt aus:

---

**Algorithmus 15** Löschen aus einem Suchbaum

---

**Prozedur** LÖSCHEN( $v, x$ )

**Falls**  $v = \text{NIL}$  **dann**

**Antworte** "nicht gefunden"

**sonst falls**  $v.D.x = x$  **dann**

**Antworte** LÖSCHEWURZEL( $v$ )

**sonst falls**  $v.D.x < x$  **dann**

$v.rechts := \text{LÖSCHEN}(v.rechts, x)$

**Antworte**  $v$

**sonst**

$v.links := \text{LÖSCHEN}(v.links, x)$

**Antworte**  $v$

**Ende Falls**

**Ende Prozedur**

---

$\triangleright v.D.x > x$

---

**Algorithmus 16** Löschen der Wurzel aus einem Suchbaum

---

**Prozedur** LÖSCHEWURZEL( $v$ )**Falls**  $v.links \neq \text{NIL}$  **dann** $(D_{\max}, v_l) := \text{EXTRAHIEREMAX}(v.links)$ Sei  $v'$  neuer Knoten $v'.D := D_{\max}$  $v'.links := v_l$  $v'.rechts := v.rechts$ **Antworte**  $v'$ **sonst falls**  $v.rechts \neq \text{NIL}$  **dann** $(D_{\min}, v_r) := \text{EXTRAHIEREMIN}(v.rechts)$ Sei  $v'$  neuer Knoten $v'.D := D_{\min}$  $v'.links := v.links$  $v'.rechts := v_r$ **Antworte**  $v'$ **sonst** $\triangleright v$  ist ein Blattknoten**Antworte** NIL**Ende Falls****Ende Prozedur**

---

---

**Algorithmus 17** Finde und lösche Minimum aus Suchbaum

---

**Prozedur** EXTRAHIEREMIN( $v$ )**Falls**  $v = \text{NIL}$  **dann****Antworte** "Minimum nicht definiert"**sonst falls**  $v.links = \text{NIL}$  **dann** $\triangleright v$  ist Minimum**Antworte**  $(v.D, v.rechts)$ **sonst** $(D_{\min}, v'_l) := \text{EXTRAHIEREMIN}(v.links)$  $v.links := v'_l$ **Antworte**  $(D_{\min}, v)$ **Ende Falls****Ende Prozedur**

---

Insgesamt erhalten wir also die folgenden Laufzeitkomplexitäten für die Implementierung eines Wörterbuchs als Suchbaum:

1.  $M.Einfügen(D)$  in Zeit  $O(h)$
2.  $M.Löschen(x)$  in Zeit  $O(h)$
3.  $M.Suche(x)$  in Zeit  $O(h)$

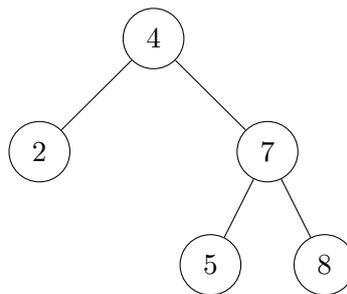
wobei  $h$  die Höhe des Baums ist. Das führt zur Frage: Was ist die Höhe des Baums?

Im schlechtesten Fall hat der Baum eine lineare Struktur. Das geschieht wenn die Elemente von  $M$  in (aufsteigend oder absteigend) sortierter Reihenfolge eingefügt werden. Dann ist  $h = n = |M|$ . Im besten Fall ist der Baum ein fast vollständiger Binärbaum, d.h. alle Schichten bis auf die letzte sind vollständig. In diesem Fall ist  $h = \Theta(\log n)$ .

Wir wollen jetzt auch (einen) Durchschnittsfall analysieren. Auch hier ist a priori nicht klar welche Wahrscheinlichkeitsverteilung betrachtet werden soll. Wir entscheiden uns für die folgende Anwendung: ein Suchbaum wird aus einer Liste  $D_1, \dots, D_n$  mit paarweise verschiedenen Schlüsseln mittels der Einfüge-Operation aufgebaut. Danach verändert er sich nicht mehr und wir suchen nach Elementen aus der Eingabeliste. Wir interessieren uns also für die durchschnittlichen Kosten einer Such-Operation. Nachdem es für diese Anwendung nur auf die Reihenfolge der Schlüssel ankommt, reicht es darauf eine Wahrscheinlichkeitsverteilung zu definieren. Ähnlich wie bei unserer Analyse von Sortieralgorithmen nehmen wir an, dass jede Permutation der Schlüssel die gleiche Wahrscheinlichkeit  $\frac{1}{n!}$  hat (Permutationsmodell). Diese Annahme hat übrigens *nicht* zur Folge dass jeder Baum mit  $n$  Knoten gleich wahrscheinlich ist, da verschiedene Permutationen den selben Baum erzeugen.

Bevor wir mit der Analyse beginnen ist es nützlich noch einige Begriffe für Bäume einzuführen: Die *Tiefe*  $d(v)$  eines Knoten  $v$  in einem Baum  $T$  ist die Anzahl der Kanten auf dem eindeutigen Pfad von  $v$  zur Wurzel von  $T$ . Die Anzahl der Schlüsselvergleiche die zum Auffinden eines Knotens  $v$  notwendig sind ist  $d(v) + 1$ . Für einen Baum  $T = (V, E)$  definieren wir die *Pfadlänge*  $L(T)$  von  $T$  durch  $L(T) = \sum_{v \in V} (d(v) + 1)$ . Die durchschnittlichen Kosten für die Suche eines in  $T$  vorhandenen Schlüssels in  $T$  belaufen sich auf  $\bar{L}(T) = \frac{L(T)}{|T|}$ .

*Beispiel 5.1.* Der Baum



hat zum Beispiel  $d(2) = 1$ ,  $d(5) = 2$ ,  $L(T) = 1 + 2 + 2 + 3 + 3 = 11$  und  $\bar{L}(T) = \frac{11}{5} = 2,2$ .

**Satz 5.1.** *Im Permutationsmodell ist der Erwartungswert der Kosten einer Such-Operation  $O(\log n)$ .*

*Beweis.* Sei  $T$  ein Baum. Falls  $T$  leer ist, dann ist  $L(T) = 0$ . Falls  $T$  nicht leer ist, sei  $T_l$  dessen linker Teilbaum und  $T_r$  dessen rechter Teilbaum. Dann ist

$$L(T) = L(T_l) + |T_l| + L(T_r) + |T_r| + 1 = L(T_l) + L(T_r) + |T|. \quad (*)$$

Dem Permutationsmodell entsprechend wählen wir uniform verteilt eine Eingabepermutation  $\pi$  von  $n$  Datensätzen mit paarweise unterschiedlichen Schlüsseln. Diese Eingabepermutation induziert einen Suchbaum  $T_\pi$  mit Pfadlänge  $L(T_\pi)$ . Wir schreiben  $EL(n)$  für den Erwartungswert der Pfadlänge von  $T_\pi$ .

Das erste Element der Eingabepermutation ist die Wurzel des Baums. Der linke Teilbaum der Wurzel wird so viele Elemente enthalten wie es  $D_i$  in der Eingabepermutation gibt deren Schlüssel kleiner als der der Wurzel ist, der rechte Teilbaum so viele Elemente wie es  $D_i$  gibt deren Schlüssel größer ist. Nachdem die Eingabepermutation zufällig war, ist für alle  $k \in \{1, \dots, n\}$  die Wahrscheinlichkeit dass das erste Element den  $k$ -t-größten Schlüssel in  $D_1, \dots, D_n$  hat  $\frac{1}{n}$ .

Mit (\*) erhalten wir also  $EL(0) = 0$  und, für  $n \geq 1$ :

$$\begin{aligned} EL(n) &= \frac{1}{n} \sum_{k=1}^n (EL(k-1) + EL(n-k) + n) \\ &= n + \frac{1}{n} \left( \sum_{k=1}^n EL(k-1) + \sum_{k=1}^n EL(n-k) \right) \\ &= n + \frac{2}{n} \sum_{k=0}^{n-1} EL(k). \end{aligned}$$

Wir haben also

$$\begin{aligned} nEL(n) &= n^2 + 2 \sum_{k=0}^{n-1} EL(k), \\ (n-1)EL(n-1) &= (n-1)^2 + 2 \sum_{k=0}^{n-2} EL(k), \\ nEL(n) - (n-1)EL(n-1) &= 2n-1 + 2EL(n-1) \text{ und} \\ EL(n) &= \frac{2n-1}{n} + \frac{n+1}{n} EL(n-1). \end{aligned}$$

Somit ist  $EL(n)$  durch eine lineare Rekursionsgleichung erster Ordnung gegeben. Also

$$\begin{aligned} EL(n) &=_{\text{Satz 4.1}} \sum_{i=1}^n \frac{2i-1}{i} \prod_{j=i+1}^n \frac{j+1}{j} = (n+1) \sum_{i=1}^n \frac{2i-1}{i(i+1)} \\ &= (n+1) \left( \sum_{i=1}^n \frac{3}{i+1} - \sum_{i=1}^n \frac{1}{i} \right) = (n+1) \left( \frac{3}{n+1} + \sum_{i=2}^n \frac{2}{i} - 1 \right) \\ &= 2(n+1)H_n - 3n \end{aligned}$$

wobei  $H_n = \sum_{i=1}^n \frac{1}{i}$  die  $n$ -te harmonische Zahl ist. Die  $n$ -te harmonische Zahl kann abgeschätzt werden durch  $H_n = \ln n + \gamma + O(\frac{1}{n})$  wobei  $\gamma$  die Euler-Mascheroni Konstante ist und einen Wert von  $0,577\dots$  hat. Wir erhalten also  $EL(n) = O(n \ln n) = O(n \log n)$  und damit  $\frac{EL(n)}{n} = O(\log n)$ .  $\square$

Suchbäume verhalten sich also (im Permutationsmodell) im Durchschnittsfall so wie im besten Fall (anders als das beim Einfügesortieren, sh. Kapitel 1, der Fall war). Die Verwendung von Suchbäumen zur Lösung des Wörterbuchproblems ist noch nicht völlig zufriedenstellend. Wir würden das Wörterbuchproblem gerne mit Operationen lösen die auch im schlechtesten Fall logarithmisch sind, umso mehr als der schlechteste Fall eine sortierte Eingabe ist (was in vielen Anwendungen mit einer Wahrscheinlichkeit von mehr als  $\frac{1}{n!}$  auftritt).

Auf Basis von Suchbäumen lässt sich auch leicht ein einfaches Sortierverfahren angeben: ein gegebenes Datenfeld  $A$  wird sortiert durch 1. einen Aufbau eines Suchbaums für  $A$  durch wiederholten Einfügen in den leeren Baum sowie 2. den Durchlauf dieses Suchbaums in symmetrischer Reihenfolge, siehe Algorithmus 13. Der zweite Schritt benötigt Laufzeit  $\Theta(n)$ . Der erste Schritt benötigt im schlechtesten Fall Laufzeit  $O(n^2)$ .

### 5.3 AVL-Bäume

In diesem Abschnitt werden wir AVL-Bäume betrachten, die auf Adelson-Velskij und Landis zurückgehen. Mit diesen ist es möglich das Wörterbuchproblem so zu lösen, dass alle drei Operationen, selbst im schlechtesten Fall, eine logarithmische Laufzeit haben. Dazu benötigen wir noch einige vorbereitende Begriffe und Beobachtungen.

Die *Höhe*  $h(v)$  eines Knotens  $v$  ist die maximale Anzahl von Knoten auf einem Pfad von  $v$  zu einem Blatt. Die Höhe des leeren Baums ist 0. Die Höhe eines nicht-leeren Baums ist die Höhe seiner Wurzel und damit  $\geq 1$ .

**Definition 5.2.** Sei  $T$  ein Suchbaum,  $v$  ein Knoten in  $T$ ,  $T_l$  der linke Teilbaum von  $v$  und  $T_r$  der rechte Teilbaum von  $v$ . Dann ist der *Balancegrad* von  $v$  definiert als  $\text{bal}(v) = h(T_r) - h(T_l)$ . Ein Suchbaum  $T$  heißt *balanciert* falls für jeden Knoten  $v$  von  $T$  gilt:  $|\text{bal}(v)| \leq 1$ .

**Satz 5.2.** *Ein balancierter Baum mit  $n$  Knoten hat Höhe  $\Theta(\log n)$ .*

*Beweis.* Ein balancierter Baum mit Höhe  $h$  und maximaler Anzahl von Knoten ist der vollständige Binärbaum. Dieser hat  $n = \Theta(2^h)$  Knoten. Für einen beliebigen balancierten Baum ist also  $h = \Omega(\log n)$ .

Ein balancierter Baum mit Höhe  $h$  und minimaler Anzahl von Knoten hat für  $h = 1$  einen Knoten und für  $h = 2$  zwei Knoten. Für  $h \geq 3$  hat er zwei Teilbäume, einen davon ein balancierter Baum mit Höhe  $h - 1$  und minimaler Anzahl von Knoten, den anderen ein balancierter Baum mit Höhe  $h - 2$  und minimaler Anzahl von Knoten. Wir erhalten also für die minimale Anzahl von Knoten  $n_h$  in einem balancierten Baum mit Höhe  $h$  die Rekursionsgleichung

$$n_h = n_{h-1} + n_{h-2} + 1 \text{ mit } n_1 = 1 \text{ und } n_2 = 2.$$

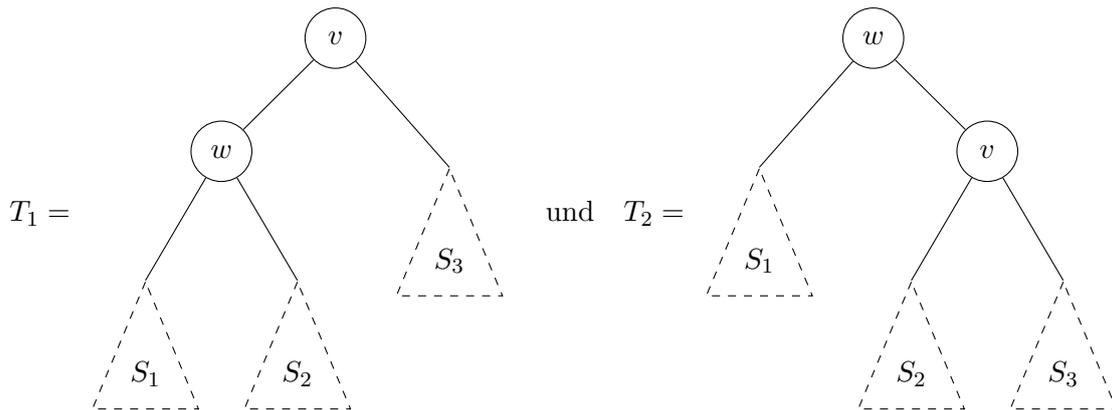
Man sieht sofort dass  $n_h \geq F_h$ . Wir wissen bereits dass

$$F_h = \frac{1}{\sqrt{5}} \left( \left( \frac{1 + \sqrt{5}}{2} \right)^h - \left( \frac{1 - \sqrt{5}}{2} \right)^h \right)$$

Nun geht  $\left( \frac{1 - \sqrt{5}}{2} \right)^h \rightarrow 0$  für  $h \rightarrow \infty$  und damit  $F_h = \Theta(\Phi^h)$  für den goldenen Schnitt  $\Phi = \frac{1 + \sqrt{5}}{2}$ . Also  $n_h = \Omega(\Phi^h)$ . Für einen beliebigen balancierten Baum gilt also  $h = O(\log_{\Phi} n) = O(\log n)$ . Insgesamt gilt also für einen beliebigen balancierten Baum  $h = \Theta(\log n)$ .  $\square$

Der wesentliche Punkt an einem AVL-Baum wird nun sein, dass er ein balancierter Suchbaum ist. Um die Balanciertheit beizubehalten müssen die Einfüge- und Löschoptionen modifiziert werden. Das wird durch Korrekturtransformationen erreicht, die sich auf die folgenden Rotationsoperationen stützen.

**Definition 5.3.** Seien

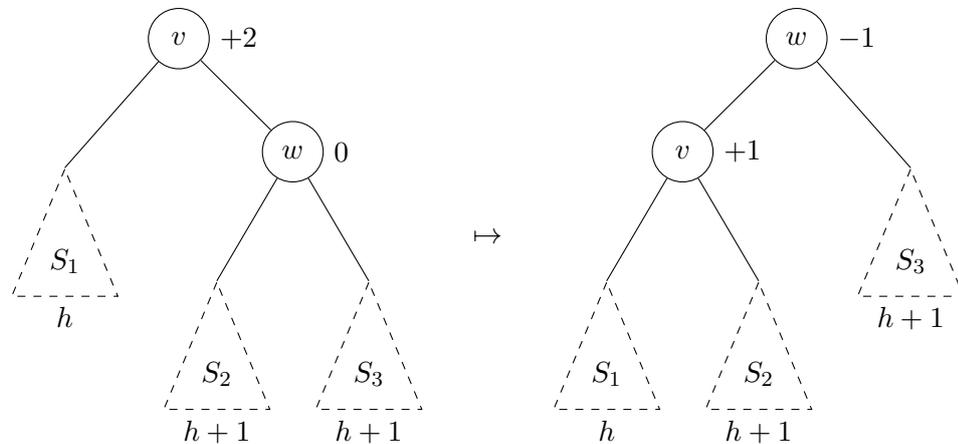


wobei  $v$  und  $w$  Knoten sind und  $S_1$ ,  $S_2$  und  $S_3$  Bäume. Dann bezeichnen wir die Abbildung von  $T_1$  auf  $T_2$  als Rechtsrotation (an der Stelle  $v$ ) und die Abbildung von  $T_2$  nach  $T_1$  als Linksrotation (an der Stelle  $w$ ).

Die Suchbaumeigenschaft wird durch diese Rotationen beibehalten. Seien nämlich  $x_1$ ,  $x_2$  und  $x_3$  Schlüssel in  $S_1$ ,  $S_2$  bzw.  $S_3$ , dann gilt  $x_1 < w.D.x < x_2 < v.D.x < x_3$  sowohl in  $T_1$  als auch in  $T_2$ .

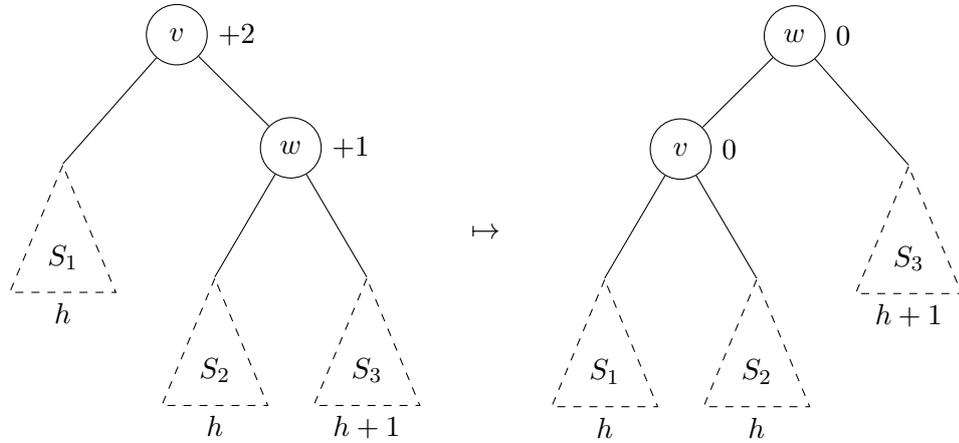
Konkret entwickeln wir die Korrekturtransformationen wie folgt: Zunächst einmal stellen wir fest, dass die Eigenschaft der Balanciertheit unabhängig von den Schlüsseln der Datensätze ist, sondern nur von der Form des Baums abhängt. Die Einfügeoperation verändert die Form des Baums indem sie ein neues Blatt hinzufügt, die Löschoption indem sie ein bestehendes Blatt entfernt. Der Balancegrad aller Knoten die nicht auf dem Pfad von der Wurzel zur Stelle der Änderung liegen bleibt also unverändert. Der Balancegrad der Knoten auf diesem Pfad verändert sich um höchstens 1. Insgesamt haben wir bei der Korrektur der Form des Baums also mit  $O(h)$  Knoten zu tun deren Balancegrad in  $\{-2, -1, 0, 1, 2\}$  liegt. Wir skizzieren eine Prozedur  $BALANCIEREN(v)$ . Die Prozedur erhält einen Baum  $T$  über dessen Wurzel  $v$  als Eingabe. Von  $T$  wird angenommen, dass  $|\text{bal}(v)| \leq 2$  und für alle Knoten  $w \in T \setminus \{v\}$  gilt:  $|\text{bal}(w)| \leq 1$ . Die Prozedur gibt einen balancierten Suchbaum mit den Elementen von  $T$  zurück.  $BALANCIEREN(v)$  geht wie folgt vor:

1. Falls  $|\text{bal}(v)| \leq 1$ , dann antworte mit  $v$ .
2. Falls  $\text{bal}(v) = +2$ , dann existiert ein Knoten  $w = v.\text{rechts}$ .
  - (a) Falls  $\text{bal}(w) = 0$ , dann führen wir die folgende Linksrotation durch



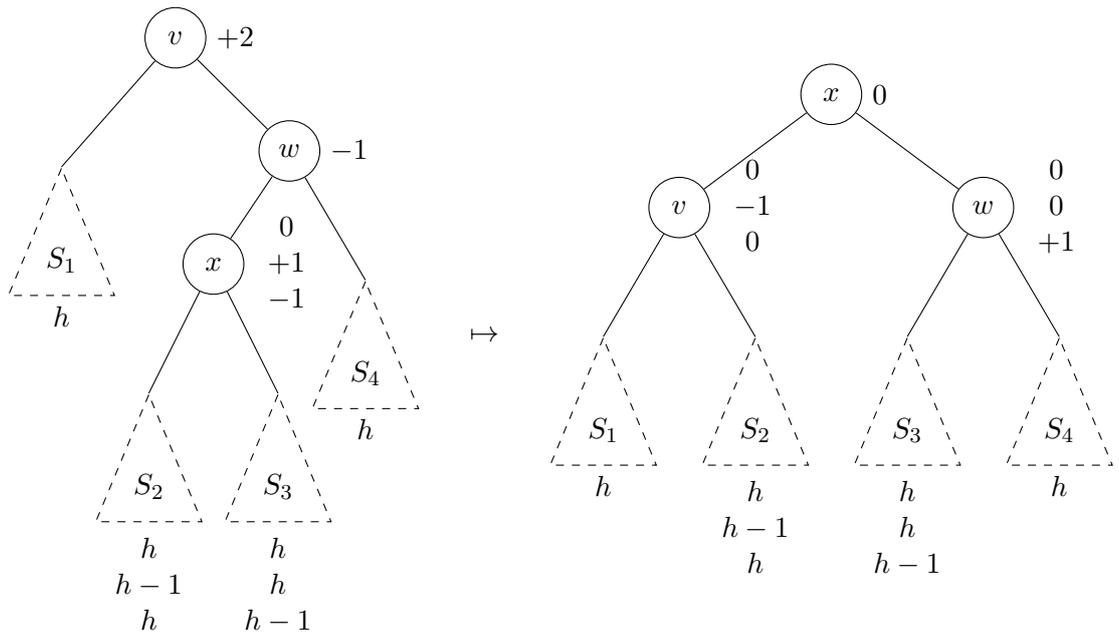
und antworten mit  $w$ . Der Eingabebaum hat Tiefe  $h + 3$ . Der Ausgabebaum hat ebenfalls Tiefe  $h + 3$ .

- (b) Falls  $\text{bal}(w) = +1$ , dann führen wir die folgende Linksrotation durch



und antworten mit  $w$ . Der Eingabebaum hat Tiefe  $h + 3$ . Der Ausgabebaum hat Tiefe  $h + 2$ .

- (c) Falls  $\text{bal}(w) = -1$ , dann existiert ein Knoten  $x = w.\text{links}$ . Wir führen die folgende Doppelrotation durch



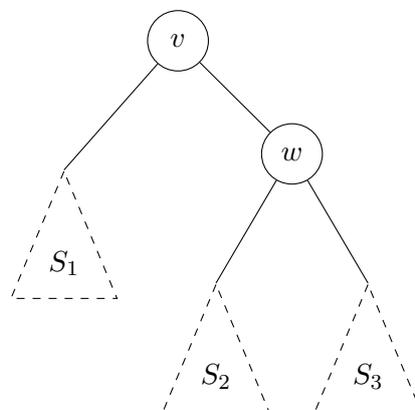
und antworten mit  $x$ . Der Eingabebaum hat Tiefe  $h + 3$ . Der Ausgabebaum hat Tiefe  $h + 2$ .

3. Der Fall  $\text{bal}(v) = -2$  ist symmetrisch zum vorherigen.

Ein AVL-Baum erlaubt nun eine effiziente Implementierung dieser Prozedur indem jeder Knoten  $v$ , zusätzlich zu  $v.D$ ,  $v.\text{links}$ ,  $v.\text{rechts}$  noch ein Feld  $v.\text{bal}$  für den Balancegrad von  $v$  hat. Der Balancegrad wird im Knoten gespeichert um die (linearen) Kosten zu seiner Bestimmung einzusparen. Damit muss der in den Knoten gespeicherte Balancegrad natürlich auch durch alle

ändernden Operationen aktualisiert werden. Erreicht ist dadurch dass die Prozedur BALANCIEREN eine Laufzeitkomplexität von  $\Theta(1)$  hat.

Die Operationen in AVL-Bäumen können dann folgendermaßen implementiert werden. Die Suchoperation in einem AVL-Baum ist wie die in einem allgemeinen Suchbaum. Die Einfüge- und Löschoptionen müssen an jeder Stelle  $v$  den neuen Balancegrad  $\text{bal}(v)$  berechnen, und, zumindest falls  $|\text{bal}(v)| = 2$ , die Prozedur BALANCIEREN( $v$ ) aufrufen. Den neuen Balancegrad auf die naive Weise zu berechnen, d.h. durch Bestimmung der Höhen der Teilbäume, ist keine Option, da dies lineare Zeit benötigen würde. Um den neuen Balancegrad in konstanter Zeit zu berechnen, muss er aus dem alten Balancegrad mit Hilfe von Informationen berechnet werden, die den Einfüge- und Löschoptionen zur Verfügung stehen. Konkret wird dazu die Veränderung der Höhe der Teilbäume benötigt. Diese beiden Operationen müssen also im rekursiven Aufstieg die Information mitführen, ob der aktuelle Teilbaum seine Höhe verändert hat, d.h. ob sie um 1 gestiegen ist (im Fall von Einfügen) oder um 1 gefallen (im Fall von Löschen). Zur Illustration geben wir hier ein Beispiel. Wir betrachten einen Baum der Form



in dem EINFÜGEN( $v, D$ ) den Aufruf EINFÜGEN( $w, D$ ) gemacht hat (d.h. der Schlüssel  $x$  des einzufügenden Elements war größer als  $v.D.x$ ). Wir gehen davon aus, dass die Prozedur EINFÜGEN so modifiziert wurde, dass sie mit einem Paar  $(v, \Delta h)$  antwortet wobei  $v$ , wie gehabt, der neue Baum ist und  $\Delta h$  die Differenz zwischen der Höhe des neuen Baums und der Höhe des alten Baums. Ebenso wird die Prozedur BALANCIEREN so modifiziert, dass sie mit dem Paar  $(v, \Delta h)$  antwortet wobei  $v$  wiederum der neue Baum ist und  $\Delta h$  die Höhenänderung vom alten zum neuen Baum. Sei nun  $(w, \Delta h_w)$  die Antwort von EINFÜGEN( $w, D$ ). Der Balancegrad von  $v$  sowie  $\Delta h_v$  berechnen sich dann wie folgt:

$$\begin{aligned} v.\text{bal} &:= v.\text{bal} + \Delta h_w \\ (v, \Delta_b) &:= \text{BALANCIEREN}(v) \\ \Delta h_v &:= \Delta h_w + \Delta_b \end{aligned}$$

An den anderen Stellen des Pseudocodes von Einfügen und Löschen werden ähnliche Änderungen durchgeführt. Diese Operationen haben damit eine Laufzeit von  $O(\log n)$  wobei  $n$  die Anzahl der Datensätze ist.

Logarithmische Komplexität ist in der Praxis sehr gering. Zum Beispiel kann ein Suchbaum der alle ca. 8 Mrd. Menschen die auf der Erde leben enthält mit einer Tiefe von ca. 33 erstellt werden. Mit AVL-Bäumen haben wir also das Wörterbuchproblem auf zufriedenstellende Weise gelöst.

## 5.4 Stapel und Warteschlangen

Ein Stapel (engl. *stack*) ist eine Datenstruktur welche die folgenden Operationen zur Verfügung stellt:

1. *Hinzufügen*( $S, D$ ) legt den Datensatz  $D$  auf den Stapel  $S$  (engl. *push*).
2. *Entfernen*( $S$ ) gibt das oberste Element vom Stapel zurück und entfernt es vom Stapel (engl. *pop*).

Dieses Prinzip bezeichnet man auch als LIFO “last in first out”. Ein Stapel kann als einfach verkettete Liste  $S$  mit einem Startzeiger  $S.Start$  implementiert werden, wobei der Stapel leer ist genau dann wenn  $S.Start = \text{NIL}$ , siehe Algorithmus 18. Diese Operationen benötigen jeweils Zeit  $\Theta(1)$ .

---

### Algorithmus 18 Stapel

---

**Prozedur HINZUFÜGEN**( $S, D$ )

Sei  $v$  neuer Knoten  
 $v.D := D$   
 $v.nächster := S.Start$   
 $S.Start := v$

**Ende Prozedur**

**Prozedur ENTFERNEN**( $S$ )

**Falls**  $S.Start = \text{NIL}$  **dann**  
    **Antworte** “Stapel leer”  
**sonst**  
     $D := S.Start.D$   
     $S.Start := S.Start.nächster$   
    **Antworte**  $D$

**Ende Falls**

**Ende Prozedur**

---

Wir wollen nun eine Warteschlange (ähnlich wie an einer Supermarktkassa) implementieren. Es soll möglich sein, das erste Element aus einer Warteschlange zu entfernen (z.B. um es abzuarbeiten) sowie ein neues Element an das Ende der Warteschlange zu stellen. Die Elemente werden also in der Reihenfolge ihres Einlangens abgearbeitet. Dieses Prinzip bezeichnet man auch als FIFO “first in first out”, die gewünschte Datenstruktur als Warteschlange (engl. *FIFO queue*). Wir wollen die folgenden Operationen

1. *Hinzufügen*( $W, D$ ) stellt  $D$  an das Ende der Warteschlange  $W$ .
2. *Entfernen*( $W$ ) gibt das Element vom Anfang der Warteschlange zurück und entfernt es.

Eine Möglichkeit eine solche Warteschlange  $W$  zu implementieren besteht darin eine einfach verkettete Liste zu verwenden mit dem Startzeiger  $W.Start$  und einem zusätzlichen Endezeiger  $W.Ende$ . Die Implementierung in Algorithmus 19 geht davon aus, dass die Warteschlange leer ist genau dann wenn der Startzeiger  $\text{NIL}$  ist. Wie man leicht sehen kann benötigen diese Operationen jeweils  $\Theta(1)$  Zeit.

---

**Algorithmus 19** Warteschlange

---

**Prozedur HINZUFÜGEN( $W, D$ )**Sei  $v$  neuer Knoten $v.D := D$  $v.nächster := \text{NIL}$ **Falls**  $W.Start = \text{NIL}$  **dann** $W.Start := v$ **sonst** $W.Ende.nächster := v$ **Ende Falls** $W.Ende := v$ **Ende Prozedur****Prozedur ENTFERNEN( $W$ )****Falls**  $W.Start = \text{NIL}$  **dann****Antworte** "Warteschlange leer"**sonst** $D := W.Start.D$  $W.Start := W.Start.nächster$ **Antworte**  $D$ **Ende Falls****Ende Prozedur**

---

## 5.5 Halden und Prioritätswarteschlangen

Oft reicht ein so einfaches Konzept einer Warteschlange aber nicht aus. Von einer Prioritätswarteschlange spricht man, wenn die Elemente nach Priorität sortiert sind. Für einen Datensatz  $D$  schreiben wir jetzt  $D.x$  für die Priorität von  $D$ . Die Priorität übernimmt also die Funktion des Schlüssels als Sortierkriterium. Wie nehmen allerdings nicht mehr an, dass die Prioritäten paarweise verschieden sind. Für eine Prioritätswarteschlange  $Q$  wollen wir zumindest die folgenden Operationen zur Verfügung stellen:

1.  $Einfügen(Q, D)$  fügt das Element  $D$  in  $Q$  (an geeigneter Stelle) ein.
2.  $Maximum(Q)$  gibt das Element maximaler Priorität zurück.
3.  $ExtrahiereMaximum(Q)$  löscht das Element maximaler Priorität aus  $Q$  und gibt es zurück.
4.  $ErhöhePriorität(Q, D, x)$  erhöht die Priorität des Elements  $D$  auf  $x$  wobei angenommen wird dass  $x \geq D.x$ .

Eine Prioritätswarteschlangen wird üblicherweise basierend auf der Datenstruktur der *Halde* (engl. *heap*) implementiert.

**Definition 5.4.** Eine Halde ist ein binärer Baum mit den folgenden Eigenschaften:

1. Alle bis auf das unterste Level sind vollständig aufgefüllt.
2. Das unterste Level ist von links bis zu einem bestimmten Punkt vollständig aufgefüllt.
3. Falls  $w$  ein Kind von  $v$  ist, dann ist  $Priorität(w) \leq Priorität(v)$ .

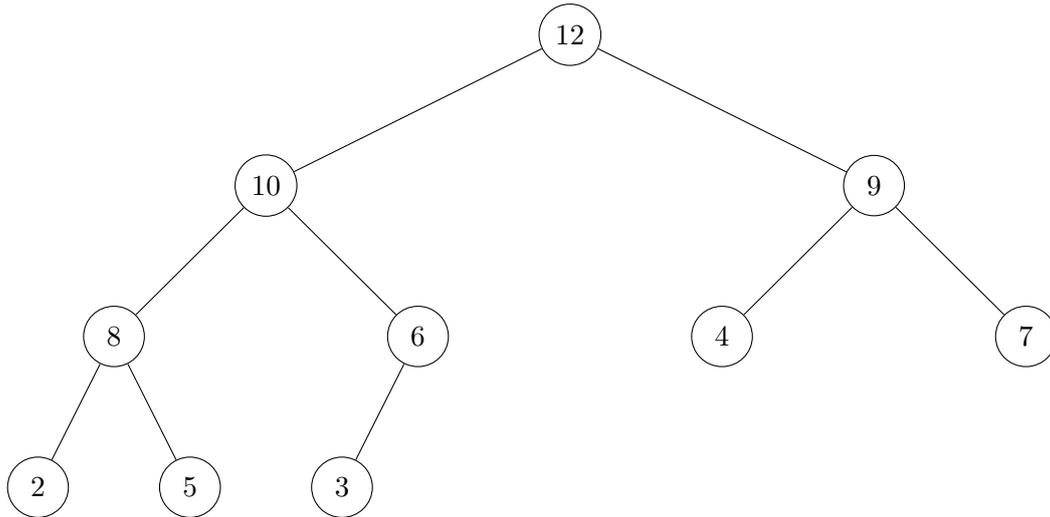


Abbildung 5.4: Eine Halde in Baumdarstellung

Daraus folgt unmittelbar dass die Wurzel des Baums maximale Priorität hat. Eine Halde kann (ähnlich wie ein Suchbaum) im Speicher mit Hilfe von Knoten abgelegt werden, die jeweils einen Zeiger *links* und einen Zeiger *rechts* haben. Wie wollen hier aber eine andere Darstellung angeben, die, je nach Anwendung, gewisse Vor- und Nachteile hat. Da alle Level bis auf das letzte vollständig ausgefüllt sind, können wir die Halde auf effiziente Weise in einem Datenfeld speichern das Level für Level von oben nach unten und, pro Level, von links nach rechts befüllt wird.

*Beispiel 5.2.* Die in Abbildung 5.4 als Baum dargestellte Halde hat als Datenfeld die folgende Form:

12	10	9	8	6	4	7	2	5	3
----	----	---	---	---	---	---	---	---	---

Um diese Darstellung einer Halde  $Q$  zu realisieren, speichern wir zusätzlich zu  $Q.Länge$  auch noch die Haldenlänge  $Q.HLänge$  die angibt bis zu welchem Punkt das Datenfeld befüllt ist. Wir werden bei allen Operationen annehmen dass  $Q.Länge$  hinreichend groß ist. In der Praxis ist das z.B. dann erfüllt wenn im Vorhinein bekannt ist, wie viele Elemente  $Q$  höchstens enthalten wird. Falls diese Anzahl nicht im Vorhinein bekannt ist, muss bei einer Einfügeoperation gegebenenfalls die Länge des zugrundeliegenden Datenfelds erhöht werden, was mit zusätzlicher Komplexität verbunden sein kann.

Wir werden Elemente der Halde mit ihrem Index im zugrundeliegenden Datenfeld referenzieren. Dann hat das linke Kind des Elements mit Index  $i$  den Index  $2i$ , das rechte Kind den Index  $2i+1$  und der Vater den Index  $\lfloor \frac{i}{2} \rfloor$ . Dementsprechend definieren wir die Prozeduren  $LINKS(i) := 2i$ ,  $RECHTS(i) := 2i+1$  und  $VATER(i) := \lfloor \frac{i}{2} \rfloor$ .

Das Einfügen eines neuen Elements geschieht dadurch, dass es zunächst an die letzte Stelle geschrieben wird und danach aufwärts an die richtige Stelle verschoben wird, siehe Algorithmus 20. Auf ähnliche Weise kann die Erhöhung der Priorität eines Elements implementiert werden, siehe Algorithmus 21. AUFWÄRTSKORRIGIEREN und damit auch EINFÜGEN und ERHÖHEPRIORITÄT benötigt in einer Halde mit  $n$  Elementen  $O(\log n)$  Zeit.

Zur Extraktion des Maximums geht man dual dazu vor: zunächst wird das Maximum (das sich ja an der Wurzel  $Q[1]$  befindet) entfernt und durch das Element am Ende der Prio-

---

**Algorithmus 20** Einfügen in eine Prioritätswarteschlange

---

**Prozedur** EINFÜGEN( $Q, D$ ) $Q.HLänge := Q.HLänge + 1$ ▷ Annahme:  $Q.Länge$  ist hinreichend groß $Q[Q.HLänge] := D$ AUFWÄRTSKORRIGIEREN( $Q, Q.HLänge$ )**Ende Prozedur****Prozedur** AUFWÄRTSKORRIGIEREN( $Q, i$ )**Solange**  $i > 1$  **und**  $Q[VATER(i)].x < Q[i].x$ Vertausche  $Q[i]$  und  $Q[VATER(i)]$  $i := VATER(i)$ **Ende Solange****Ende Prozedur**

---

---

**Algorithmus 21** Erhöhung der Priorität in einer Prioritätswarteschlange

---

**Vorbedingung:**  $x \geq Q[i].x$ **Prozedur** ERHÖHEPRIORITÄT( $Q, i, x$ ) $Q[i].x := x$ AUFWÄRTSKORRIGIEREN( $Q, i$ )**Ende Prozedur**

---

ritätswarteschlange ersetzt. Danach wird dieses Element, das ja jetzt zu niedrige Priorität für seine Position hat, Stück für Stück durch Vertauschungen an eine korrekte Stelle (abwärts) verschoben, siehe Algorithmus 22. ABWÄRTSKORRIGIEREN und damit auch EXTRAHIEREMAXIMUM benötigt ebenfalls  $O(\log n)$  Zeit.

---

**Algorithmus 22** Extraktion des Maximums aus Prioritätswarteschlange

---

**Prozedur** EXTRAHIEREMAXIMUM( $Q$ )**Falls**  $Q.HLänge = 0$  **dann****Antworte** “ $Q$  ist leer”**sonst** $max := Q[1]$  $Q[1] := Q[Q.HLänge]$  $Q.HLänge := Q.HLänge - 1$ ABWÄRTSKORRIGIEREN( $Q, 1$ )**Antworte**  $max$ **Ende Falls****Ende Prozedur****Prozedur** ABWÄRTSKORRIGIEREN( $Q, i$ )**Solange**  $i \leq Q.HLänge$ Sei  $m \in \{i, \text{LINKS}(i), \text{RECHTS}(i)\} \cap [1, \dots, Q.HLänge]$  so dass  $Q[m].x$  maximal ist**Falls**  $m = i$  **dann**

▷ Abwärts-Korrektur beendet

 $i := Q.HLänge + 1$ **sonst**Vertausche  $Q[i]$  und  $Q[m]$  $i := m$ **Ende Falls****Ende Solange****Ende Prozedur**

---

Anders als bei den Operationen für AVL-Bäume, die funktional und rekursiv implementiert waren, haben wir uns bei diesen Korrekturoperationen für eine imperative Implementierung entschieden. Der funktionale Programmierstil harmoniert üblicherweise gut mit Bäumen (und sonstigen rekursiv definierten Datenstrukturen), der imperative mit Datenfeldern (und anderen Datenstrukturen mit globalem Zugriff).

Auf Basis von Halden lässt sich auch ein Sortierverfahren angeben: *Haldensortieren* (engl. *heapsort*). Die Grundidee dieses Verfahrens besteht darin aus dem Eingabedatenfeld eine Halde aufzubauen und dann Schritt für Schritt jeweils das Maximum der Halde zu entfernen und an das Ende des Ausgabedatenfelds zu schreiben. Aufgrund der gewählten Implementierung einer Halde als Datenfeld können wir in diesem Verfahren für das Eingabedatenfeld, die Halde und das Ausgabedatenfeld den selben Speicherbereich benutzen, siehe Algorithmus 23. Man sieht sofort ein, dass ERZUEGEHALDE in Zeit  $O(n \log n)$  läuft. Tatsächlich benötigt diese Prozedur sogar nur Zeit  $O(n)$ : Eine Halde mit  $n$  Elementen hat Höhe  $\lfloor \log n \rfloor$  und höchstens  $\lceil \frac{n}{2^{h+1}} \rceil$  Knoten der Höhe  $h$ . ABWÄRTSKORRIGIEREN( $A, i$ ) benötigt Zeit  $O(h)$  wobei  $h$  die Höhe von  $A[i]$  ist. Insgesamt erhalten wir also für die Laufzeit von ERZUEGEHALDE:

$$\sum_{h=0}^{\lfloor \log n \rfloor - 1} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor - 1} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h})$$

---

**Algorithmus 23** Haldensortieren (engl. *heapsort*)

---

**Prozedur** HALDENSORTIEREN( $A$ )  
  ERZEUGEHALDE( $A$ )  
  **Für**  $i := A.Länge, \dots, 2$   
    Vertausche  $A[1]$  und  $A[i]$   
     $A.HLänge := A.HLänge - 1$   
    ABWÄRTSKORRIGIEREN( $A, 1$ )  
  **Ende Für**  
**Ende Prozedur**

**Prozedur** ERZEUGEHALDE( $A$ )  
   $A.HLänge := A.Länge$   
  **Für**  $i := \text{VATER}(A.HLänge), \dots, 1$   
    ABWÄRTSKORRIGIEREN( $A, i$ )  
  **Ende Für**  
**Ende Prozedur**

---

Aus  $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$  erhält man durch Differenzieren und Multiplikation mit  $x$  die Gleichung  $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  und damit für  $x = \frac{1}{2}$ :

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

Insgesamt benötigt HALDENSORTIEREN dann Zeit  $O(n \log n)$ .

Statt einer Sortierung nach maximaler Priorität kann natürlich auch eine Prioritätswarteschlange mit Sortierung nach minimaler Priorität auf symmetrische Weise implementiert werden. Die hier beschriebenen Datenstrukturen heißen in der Literatur auch *max-priority queue* basierend auf *max-heaps*, die symmetrischen *min-priority queue* basierend auf *min-heaps*.



# Kapitel 6

## Suchen und Sortieren in Graphen

### 6.1 Breitensuche und Tiefensuche

Wenn wir einen Baum vollständig durchlaufen wollen, dann kann das unter anderem auf die folgenden beiden Arten bewerkstelligt werden: 1. Schicht für Schicht, indem wir also möglichst spät weiter absteigen und 2. indem wir möglichst früh weiter absteigen. Die erste Vorgehensweise wird auch als Breitensuche (engl. *breadth-first search (BFS)*) bezeichnet, die zweite als Tiefensuche (engl. *depth-first search (DFS)*), siehe Abbildung 6.1. Analoge Vorgehensweisen sind auch in beliebigen Graphen möglich.

Der Ansatz zur Breitensuche in einem beliebigen Graphen besteht darin, von einem Startknoten  $u$  ausgehend den Graphen in alle Richtungen zu durchlaufen wobei die Knoten Ebene für Ebene abgearbeitet werden. Alle neu entdeckten Knoten werden in einer Warteschlange zur späteren Bearbeitung eingereiht. Dabei führen wir eine Prozedur  $P$  auf allen gefundenen Knoten aus, siehe Algorithmus 24.

Für die folgende Laufzeitanalyse wollen wir davon ausgehen, dass der Eingabegraph durch eine Adjazenzliste gegeben ist und dass die Prozedur  $P$  nur konstante Laufzeit benötigt. Wie bereits in Abschnitt 5.4 festgestellt, benötigen HINZUFÜGEN und ENTFERNEN jeweils  $\Theta(1)$  Zeit. Jeder Knoten der von  $u$  aus erreichbar ist wird genau ein Mal in die Warteschlange eingereiht und genau ein Mal aus ihr entnommen, also finden insgesamt  $O(|V|)$  viele Warteschlangen-Operationen statt. Für jeden erreichten Knoten wird genau ein Mal die Liste seiner ausgehenden

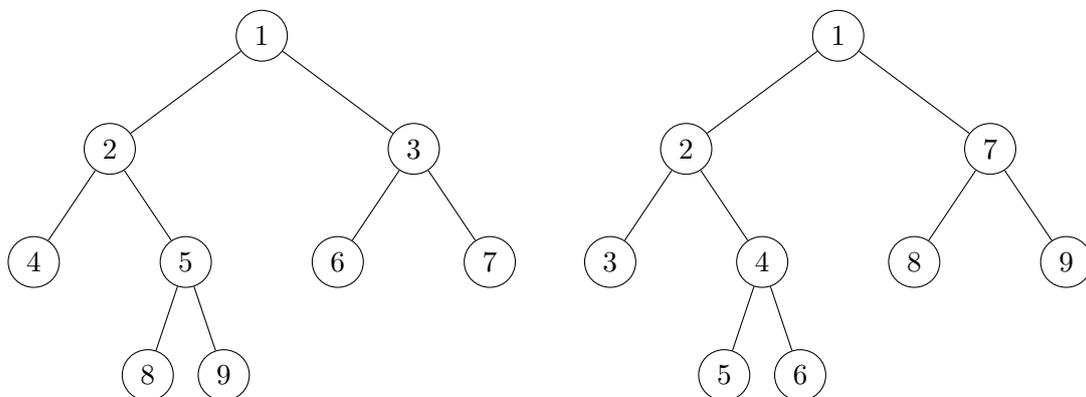


Abbildung 6.1: Breitensuche (links) und Tiefensuche (rechts) in einem Baum

---

**Algorithmus 24** Breitensuche (engl. *breadth-first search*)

---

**Prozedur** BREITENSUCHE( $V, E, u$ )

Sei *bekannt* neues Datenfeld der Länge  $|V|$ , überall mit **falsch** initialisiert

Sei  $Q$  eine neue Warteschlange, leer initialisiert

$bekannt[u] := \mathbf{wahr}$

HINZUFÜGEN( $Q, u$ )

**Solange**  $Q$  nicht leer

$v := \text{ENTFERNEN}(Q)$

    P( $v$ )

**Für** jede Kante  $(v, w) \in E$

**Falls**  $bekannt[w] = \mathbf{falsch}$  **dann**

$bekannt[w] = \mathbf{wahr}$

            HINZUFÜGEN( $Q, w$ )

**Ende Falls**

**Ende Für**

**Ende Solange**

**Ende Prozedur**

---

Kanten durchlaufen, somit wird also die innerste Schleife  $O(|E|)$  mal durchlaufen. Insgesamt erhalten wir also die Laufzeit  $O(|V| + |E|)$ . Dieser Algorithmus ruft die Prozedur P auf allen von  $u$  aus erreichbaren Knoten auf. Knoten die von  $u$  aus nicht erreicht werden können werden nicht besucht.

Die Tiefensuche kann nun, bis auf die Reihenfolge der Knoten, auf die gleiche Weise organisiert werden. Die für die Tiefensuche gewünschte Reihenfolge erhalten wir, indem wir die Warteschlange (FIFO) durch einen Stapel (LIFO) ersetzen. Wie vorhin führen wir auch jetzt eine Prozedur P auf genau den von  $u$  aus erreichbaren Knoten aus, siehe Algorithmus 25. Alles was

---

**Algorithmus 25** Tiefensuche (engl. *depth-first search*)

---

**Prozedur** TIEFENSUCHE( $V, E, u$ )

Sei *bekannt* neues Datenfeld der Länge  $|V|$ , überall mit **falsch** initialisiert

Sei  $S$  ein neuer Stapel, leer initialisiert

$bekannt[u] := \mathbf{wahr}$

HINZUFÜGEN( $S, u$ )

**Solange**  $S$  nicht leer

$v := \text{ENTFERNEN}(S)$

    P( $v$ )

**Für** jede Kante  $(v, w) \in E$

**Falls**  $bekannt[w] = \mathbf{falsch}$  **dann**

$bekannt[w] = \mathbf{wahr}$

            HINZUFÜGEN( $S, w$ )

**Ende Falls**

**Ende Für**

**Ende Solange**

**Ende Prozedur**

---

wir vorhin über die Laufzeit gesagt haben gilt auch hier. Die Tiefensuche hat also ebenfalls Laufzeit  $O(|V| + |E|)$ .

*Beispiel 6.1.* siehe `bsp.breitensuche.tiefensuche.pdf`.

Breitensuche und Tiefensuche kann nun für eine bestimmte Prozedur  $P$ , wie in Algorithmen 24 und 25 angegeben, direkt verwendet werden. Größere Bedeutung als diese Prozeduren im Wortlaut hat aber das Designprinzip, einen Graphen entlang seiner Kanten zu durchlaufen wobei problemspezifische zusätzliche Information (wie z.B. das *bekannt*-Datenfeld) mitgeführt wird. In diesem Sinn bilden die Breitensuche und Tiefensuche eine wichtige Grundlage für viele ähnliche Algorithmen.

So kann zum Beispiel durch eine ganz einfache Modifikation etwa der Tiefensuche ein Algorithmus zur Detektion von Zyklen angegeben werden: Man kann sich leicht überlegen dass ein zusammenhängender (ungerichteter) Graph  $G$  genau dann einen Zyklus enthält wenn die Breitensuche oder Tiefensuche einen Knoten erreicht, der bereits bekannt ist. Einen Algorithmus zur Detektion eines Zyklus in einem Graphen erhält man also durch Ersetzung des Körpers der innersten Schleife durch:

```

Falls bekannt( $w$ ) = wahr dann
    Antworte "Graph enthält Zyklus!"
sonst
    bekannt( $v$ ) = wahr
    HINZUFÜGEN( $S, w$ )
Ende Falls

```

## 6.2 Topologisches Sortieren

Ein weiteres Beispiel für ein Problem das durch geeignetes Durchlaufen eines Graphen gelöst werden kann ist die Erstellung einer Linearisierung eines gerichteten Graphen (in der Literatur oft auch als topologisches Sortieren bezeichnet).

**Definition 6.1.** Sei  $G = (V, E)$  ein gerichteter endlicher Graph und  $n = |V|$ . Eine *Linearisierung* von  $G$  ist eine endliche Folge  $v_1, \dots, v_n$  von Knoten so dass  $i \neq j \Rightarrow v_i \neq v_j$  und  $(v_i, v_j) \in E \Rightarrow i < j$ . Ein *gerichteter Zyklus* in  $G$  ist ein gerichteter Pfad  $v_1, \dots, v_k$  mit  $k \geq 2$  und  $(v_k, v_1) \in E$ .

*Beispiel 6.2.* Wenn wir in einem Binärbaum als gerichteten Graphen auffassen, indem die Kanten von oben nach unten orientiert werden, dann ist die Datenfeld-Darstellung einer Halde eine topologische Sortierung der Baumdarstellung der Halde.

**Satz 6.1.** Sei  $G = (V, E)$  ein endlicher gerichteter Graph. Dann hat  $G$  eine Linearisierung genau dann wenn  $G$  zyklensfrei ist.

*Beweis.* Angenommen  $G$  hat eine Linearisierung  $v_1, \dots, v_n$  und einen Zyklus. Dann muss der Zyklus die Form  $v_{i_1}, \dots, v_{i_k}$  mit  $(v_{i_k}, v_{i_1}) \in E$  haben. Damit gilt  $i_1 < i_2 < \dots < i_k < i_1$ , Widerspruch. Jeder Graph der eine Linearisierung hat muss also zyklensfrei sein.

Bevor wir die Gegenrichtung beweisen, beobachten wir dass ein endlicher gerichteter zyklensfreier Graph einen Knoten  $v$  mit  $d^-(v) = 0$  enthält: dieser kann gefunden werden, indem wir mit einem beliebigen Knoten  $v_0$  starten und eine beliebige eingehende Kante rückwärts gehen. Dieser Schritt wird solange wiederholt bis wir an einem Knoten ohne eingehende Kanten angelangt sind. Dieser Prozess terminiert, da der Graph endlich und zyklensfrei ist.

Nun zeigen wir mit Induktion nach  $|V|$  dass ein zyklensfreier Graph  $G = (V, E)$  eine Linearisierung hat. Die Induktionsbasis  $|V| = 0$  ist trivial. Für den Induktionsschritt sei  $v \in V$  mit  $d^-(v) = 0$ . Sei  $G' = G - v$ , also der Graph der aus  $G$  entsteht wenn  $v$  und alle Kanten die  $v$  enthalten entfernt worden sind. Dann hat  $G'$  nach Induktionshypothese eine Linearisierung

$v_2, \dots, v_n$ . Nachdem es kein  $u \in V \setminus \{v\}$  gibt mit  $(u, v) \in E$ , ist  $v, v_2, \dots, v_n$  eine Linearisierung von  $G$ .  $\square$

Wir wollen nun das folgende Berechnungsproblem betrachten:

**Linearisierung (Topologisches Sortieren)**

Eingabe: Ein endlicher zyklensfreier Graph  $G = (V, E)$

Ausgabe: Eine Linearisierung von  $G$

Als Anwendung kann man sich zum Beispiel Folgendes vorstellen: die Knoten repräsentieren die Teilaufgaben eines Projekts, von Aufgabe  $A$  nach Aufgabe  $B$  wird eine Kante gesetzt falls  $A$  vor  $B$  erledigt werden muss. Gesucht ist dann eine Reihenfolge in der diese Aufgaben unter Beachtung ihrer Abhängigkeiten abgearbeitet werden können.

Der Beweis von Satz 6.1 suggeriert ja bereits die folgende algorithmische Vorgehensweise:

Wiederhole  $|V|$  mal:

bestimme einen Knoten  $v$  mit  $d^-(v) = 0$ , gib  $v$  aus und entferne  $v$  aus dem Graphen.

Die Bestimmung eines Knoten  $v$  mit  $d^-(v) = 0$  kann, wie im Beweis von Satz 6.1, so gemacht werden, dass von einem beliebigen Knoten  $u$  ausgehend die Kanten zurück verfolgt werden bis ein solcher Knoten  $v$  gefunden werden wird. Wie im Beweis von Satz 6.1 garantiert die Zyklensfreiheit des Graphen dass dieser Algorithmus terminiert. Diese Methode zur Bestimmung eines  $v \in V$  mit  $d^-(v) = 0$  benötigt Zeit  $O(|V|)$ . Da sie  $|V|$  mal wiederholt wird ist der Zeitbedarf dieses Algorithmus nur mit  $O(|V|^2)$  abschätzbar.

Für dünn besetzte Graphen ist das nicht besonders gut. Eine bessere Vorgehensweise besteht darin, inspiriert von Breitensuche und Tiefensuche, den Graphen entlang seiner Kanten zu durchlaufen, wobei als nächste Knoten nur solche mit Eingangsgrad 0 (bezüglich des noch nicht durchlaufenen Teils) in Frage kommen. Um diese Knoten zu kennen wird statt dem *bekannt*-Datenfeld ein Datenfeld *Grad* mitgeführt in dem der Eingangsgrad aller Knoten bezüglich des noch nicht durchlaufenen Teils abgelegt (und auf dem aktuellen Stand gehalten) wird, siehe Algorithmus 26. Analog zur Laufzeitabschätzung der Breitensuche und Tiefensuche kann man sich auch hier leicht überlegen, dass die Laufzeit dieses Algorithmus  $\Theta(|V| + |E|)$  ist.

---

**Algorithmus 26** Linearisieren (Topologisches Sortieren)

---

**Prozedur** LINEARISIEREN( $V, E$ )

Sei  $Grad$  ein neues Datenfeld der Länge  $|V|$ , überall mit 0 initialisiert

**Für** alle  $(v, w) \in E$

$Grad[w] := Grad[w] + 1$

**Ende Für**

Sei  $M$  neue Warteschlange oder Stapel, leer initialisiert

**Für** alle  $v \in V$

**Falls**  $Grad[v] = 0$  **dann**

HINZUFÜGEN( $M, v$ )

**Ende Falls**

**Ende Für**

**Solange**  $M$  nicht leer

$v :=$  ENTFERNEN( $M$ )

AUSGABE( $v$ )

**Für** alle  $(v, w) \in E$

$Grad[w] := Grad[w] - 1$

**Falls**  $Grad[w] = 0$  **dann**

HINZUFÜGEN( $M, w$ )

**Ende Falls**

**Ende Für**

**Ende Solange**

**Ende Prozedur**

---



# Kapitel 7

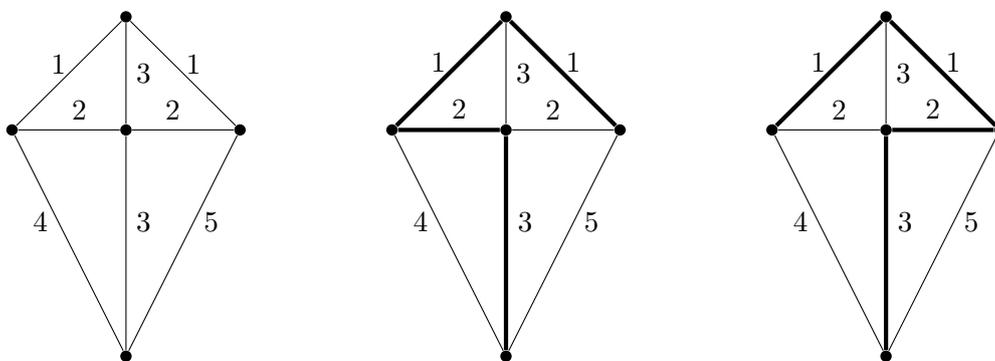
## Gierige Algorithmen

Ein gieriger Algorithmus stellt die Lösung für ein Problem dadurch zusammen dass er zu jedem Zeitpunkt der Lösung jenen Teil hinzufügt der zu diesem Zeitpunkt am vielversprechendsten ist. Einmal gemachte Entscheidungen werden dabei nicht mehr revidiert. Auf diese Weise werden lokal optimale Bausteine zu einer Lösung zusammengefügt. Für gewisse Probleme wird dadurch eine global optimale Lösung erreicht. Wenn gierige Algorithmen auch nicht immer optimale Lösungen liefern, so haben sie doch den Vorteil dass sie schnell (im Sinne der Laufzeit) und einfach (im Sinne des Implementierungsaufwands) sind.

### 7.1 Der Algorithmus von Kruskal

Ein gutes Beispiel für einen gierigen Algorithmus is der Algorithmus von Kruskal zur Berechnung eines minimalen Spannbaums eines zusammenhängenden endlichen Graphen, vgl. Abschnitt 2.3. Dieser Algorithmus ist leicht erklärt: Starte mit einer leeren Menge von Kanten  $B$  und wiederhole den folgenden Schritt so oft wie möglich: füge von den noch nicht betrachteten Kanten des Eingabegraphen  $G$  eine mit minimalen Kosten zu  $B$  hinzu falls dadurch kein Zyklus entsteht. Dieser Algorithmus vereinigt also sukzessive Bäume zu Bäumen, begonnen bei Knoten-Singletons bis daraus ein Spannbaum entsteht. Es handelt sich um einen gierigen Algorithmus da in jedem Schritt ein lokal optimales Element der Lösung (eine Kante mit minimalen Kosten) hinzugefügt wird ohne sich darum zu kümmern, ob dadurch eine global optimale Lösung entsteht (ein minimaler Spannbaum).

*Beispiel 7.1.* Ein Graph mit Kantenkosten die zwei minimale Spannäume erlauben:



**Satz 7.1.** *Der Algorithmus von Kruskal ist korrekt.*

Wir wollen jetzt einen Korrektheitsbeweis des Algorithmus von Kruskal angeben. Der Algorithmus geht wie folgt vor: sei  $e_1, \dots, e_n$  eine Sortierung von  $E$  so dass  $c(e_1) \leq \dots \leq c(e_n)$ , sei  $E_0 = \emptyset$  und

$$E_{i+1} = \begin{cases} E_i \cup \{e_{i+1}\} & \text{falls } (V, E_i \cup \{e_{i+1}\}) \text{ zyklensfrei ist und} \\ E_i & \text{sonst.} \end{cases}$$

Der Algorithmus antwortet mit  $(V, E_n)$ .

**Satz 7.2.** *Der Algorithmus von Kruskal ist korrekt.*

*Beweis.* Zunächst ist klar dass  $B = (V, E_n)$  zyklensfrei ist.  $B$  ist aber auch zusammenhängend: Angenommen  $B$  wäre nicht zusammenhängend, seien dann  $v, w \in V$  in unterschiedlichen Zusammenhangskomponenten von  $B$ . Dann gibt es einen Pfad  $p$  von  $v$  nach  $w$  in  $G$  und eine Kante  $e_i$  auf  $p$  mit der die Zusammenhangskomponente von  $v$  verlassen wird und damit auch  $e_i \notin E_n$ . Dann ist aber  $(V, E_n \cup \{e_i\})$  zyklensfrei und damit auch  $(V, E_{i-1} \cup \{e_i\})$  zyklensfrei, also  $e_i \in E_i \subseteq E_n$ , Widerspruch.  $B$  ist also ein Spannbaum von  $G$ .

Für die Minimalität von  $B$  reicht es zu zeigen dass für alle  $i \in \{0, \dots, n\}$  ein minimaler Spannbaum von  $G$  mit Kantenmenge  $M_i$  existiert so dass  $E_i \subseteq M_i$ . Dann ist nämlich  $E_n = M_n$  und damit  $B$  minimal. Wir gehen mit Induktion nach  $i$  vor. Der Fall  $i = 0$  ist trivial. Für den Induktionsschritt definieren wir  $M_{i+1} = M_i$  falls keine Kante hinzugefügt wird oder die hinzugefügte Kante  $e_{i+1} \in M_i$  ist. Sei nun also  $E_{i+1} = E_i \cup \{e_{i+1}\}$ ,  $(V, E_{i+1})$  zyklensfrei und  $e_{i+1} \notin M_i$ . Dann enthält  $(V, M_i \cup \{e_{i+1}\})$  einen Zyklus. Da aber  $(V, E_{i+1})$  zyklensfrei ist, existiert ein  $e_j$  auf dem Zyklus mit  $e_j \notin E_{i+1}$ . Sei nun  $M_{i+1} = (M_i \setminus \{e_j\}) \cup \{e_{i+1}\}$ . Dann ist  $(V, M_{i+1})$  zusammenhängend da  $(V, M_i)$  zusammenhängend ist und in jedem  $M_i$ -Pfad die Kante  $e_j$  durch  $e_{i+1}$  und den Rest des Zyklus ersetzt werden kann. Weiters ist  $|M_{i+1}| = |M_i| = |V| - 1$  da ja  $e_j \in M_i$  und  $e_{i+1} \notin M_i$  ist und  $(V, M_{i+1})$  mit Satz 2.3 also ein Spannbaum von  $G$ . Außerdem ist  $c(e_j) \geq c(e_{i+1})$  denn  $c(e_j) < c(e_{i+1})$  impliziert  $j < i + 1$  sowie  $(V, E_j \cup \{e_j\})$  zyklensfrei und damit  $e_j \in E_i$ . Also ist  $c(M_{i+1}) \leq c(M_i)$  und, da  $M_i$  minimal ist,  $c(M_{i+1}) = c(M_i)$  und  $M_{i+1}$  also ebenfalls ein minimaler Spannbaum.  $\square$

Um die Laufzeit des Algorithmus von Kruskal zu analysieren formulieren wir ihn zunächst im Detail aus. Im Prinzip wäre es möglich für jede Kante einen expliziten Test auf die Zyklensfreiheit des Graphen (z.B. mit Breiten- oder Tiefensuche) durchzuführen, um zu entscheiden, ob diese Kante zum Wald hinzugefügt werden soll. Allerdings ist es geschickter, explizite Tests auf Zyklensfreiheit zu unterlassen da diese viel Zeit benötigen. Stattdessen wollen wir die Zusammenhangskomponenten des Waldes mitführen und auf dem aktuellen Stand halten. Der Algorithmus von Kruskal erzeugt einen Spannbaum durch sukzessive Vereinigung zweier Bäume in einem Wald zu einem neuen Baum durch Hinzufügen einer Kante. Er benötigt also eine Datenstruktur zur Darstellung einer Partition einer endlichen Menge, die Partition der Knoten in Zusammenhangskomponenten. Eine solche Datenstruktur zur Darstellung einer Partition  $P$  einer endlichen Menge  $M$  wird auch als *union-find*-Datenstruktur bezeichnet. Sie stellt (zumindest) die folgenden Operationen zur Verfügung:

1. INITIALISIEREN( $M$ ) erzeugt Datenstruktur für die Partition einer Menge  $M$  die aus Singleton-Klassen besteht.
2. VEREINIGEN( $P, x_1, x_2$ ) für zwei Elemente  $x_1$  und  $x_2$  der Partition  $P$ .
3. FINDEN( $P, x$ ) gibt für  $x \in M$  die Klasse  $C$  von  $P$  zurück die  $x$  enthält.

---

**Algorithmus 27** Algorithmus von Kruskal

---

**Prozedur** KRUSKAL( $V, E, c$ )  
   $B := \emptyset$   
   $P := \text{INITIALISIEREN}(V)$   
  **Für** alle  $(v, w) \in E$  in nichtfallender Reihenfolge  
    **Falls** FINDEN( $P, v$ )  $\neq$  FINDEN( $P, w$ ) **dann**  
       $B := B \cup \{v, w\}$   
       $P := \text{VEREINIGEN}(P, v, w)$   
    **Ende Falls**  
  **Ende Für**  
  **Antworte** ( $V, B$ )  
**Ende Prozedur**

---

Auf Basis einer solchen Datenstruktur für eine Partition kann der Algorithmus von Kruskal dann wie in Algorithmus 27 ausformuliert werden.

Eine einfache Implementierung der Partitionsdatenstruktur kann wie folgt erreicht werden. Sei  $M = \{1, \dots, n\}$ . Wir verwenden auch zur Bezeichnung der Äquivalenzklassen Zahlen von 1 bis  $n$ . Wir arbeiten mit einem Datenfeld *ElementInKlasse* der Länge  $n$  wobei *ElementInKlasse*[ $i$ ] die Klasse ist in der sich das Element  $i$  befindet. Wir benötigen weiters ein Datenfeld *Kardinalität* der Länge  $n$  wobei *Kardinalität*[ $i$ ] die Kardinalität der Klasse  $i$  ist. Und schließlich verwenden wir noch ein weiteres Datenfeld *Elemente* der Länge  $m$  wobei *Elemente*[ $i$ ] eine einfach verkettete Liste der Elemente der Klasse  $i$  ist.

*Beispiel 7.2.* Sei  $M = \{1, \dots, 8\}$ . Nach Initialisierung und den Aufrufen  $\text{VEREINIGEN}(P, 2, 4)$ ,  $\text{VEREINIGEN}(P, 6, 8)$  und  $\text{VEREINIGEN}(P, 1, 4)$  haben die Datenfelder den folgenden Inhalt:

*ElementInKlasse:* 2, 2, 3, 2, 5, 6, 7, 6

*Kardinalität:* 0, 3, 1, 0, 1, 2, 1, 0

*Elemente:* NIL; 2, 4, 1; 3; NIL; 5; 6, 8; 7; NIL

Die Initialisierung benötigt Zeit  $\Theta(n)$ . Das Finden eines Elements geschieht direkt durch Nachsehen im Datenfeld *ElementInKlasse* und benötigt daher nur Zeit  $\Theta(1)$ . Für das Vereinigen der Klassen von  $x_1$  und  $x_2$  gehen wir wie folgt vor: zunächst seien  $C_1$  die Klasse von  $x_1$  und  $C_2$  die Klasse von  $x_2$ . Dann kann über *Kardinalität* festgestellt werden welche der beiden Klassen die größere ist, sei o.B.d.A.  $C_1$  die größere und  $C_2$  die kleinere. Wir entfernen dann die Klasse  $C_2$  und fügen alle ihre Elemente der Klasse  $C_1$  hinzu. Die dementsprechende Aktualisierung der Datenfelder benötigt Zeit  $\Theta(|C_2|)$  da die zu ändernden Indices im Datenfeld *Elemente* zu finden sind.

Der Algorithmus von Kruskal führt  $|V| - 1$  Vereinigungen aus. Jeder der Klassen hat höchstens die Größe  $|V|$ . Eine naive obere Schranke ist also  $O(|V|^2)$  für die Laufzeit der Vereinigungsoperationen. Diese Schranke kann durch eine genauere Analyse allerdings verbessert werden. Sie ist nämlich insofern nicht realistisch als sie zwei miteinander inkompatible Annahmen über den schlechtesten Fall kombiniert, nämlich 1. dass viele Vereinigungen stattfinden und 2. dass diese Vereinigungen großer Mengen sind.

**Lemma 7.1.** *In der Datenfeld-Implementierung der Partitionsdatenstruktur benötigen  $k$  sukzessive Vereinigungsoperationen beginnend bei der Singleton-Partition höchstens  $O(k \log k)$  Zeit.*

*Beweis.* In  $k$  (binären) Vereinigungsoperationen sind höchstens  $2k$  Elemente von  $M$  involviert. Sei  $i \in M$ . Wir schreiben  $C_j(i)$  für den Index der Klasse von  $i$  nach der  $j$ -ten Vereinigungs-

operation und  $|C_j(i)|$  für die Kardinalität dieser Klasse. Dann ist  $|C_k(i)| \leq 2k$ .<sup>1</sup> Außerdem gilt  $C_{j+1}(i) \neq C_j(i) \Rightarrow |C_{j+1}(i)| \geq 2|C_j(i)|$  da ja bei jeder Vereinigungsoperation der Index der größeren Klasse behalten wird. Deshalb kann sich also die Klassenzugehörigkeit von  $i$  höchstens  $\log 2k$  mal ändern. Da in  $k$  Vereinigungen höchstens  $2k$  Elemente involviert sind, erhalten wir also für die gesamte Laufzeit  $O(2k \log 2k) = O(k \log k)$ .  $\square$

In einer Situation wie der des obigen Lemmas sagen wir auch dass eine einzelne von  $k$  Vereinigungsoperation *amortisierte Kosten*  $\frac{1}{k}O(k \log k) = O(\log k)$  hat. Wir sprechen dann auch von *amortisierter Laufzeitanalyse*. Eine solche Vorgehensweise zur Abschätzung des schlechtesten Falls ist immer dann sinnvoll wenn mehrere Operationen durchgeführt werden von denen zwar eine einzelne recht teuer sein kann, die aber insgesamt im Durchschnitt nicht sehr teuer sind. In einer ähnlichen Situation waren wir auch bei der Abschätzung der Laufzeit der Erzeugung einer Halde aus einem Datenfeld im Kontext von Haldensortieren.

Die Gesamtlaufzeit des Algorithmus von Kruskal setzt sich also wie folgt zusammen: Wir benötigen  $O(|V|)$  Zeit für die Initialisierung der Partitionsdatenstruktur,  $O(|E| \log |E|)$  Zeit für das Sortieren der Kanten,  $O(|E|)$  Schleifendurchläufe sowie  $O(|V| \log |V|)$  für die  $|V| - 1$  Vereinigungsoperationen. Da in einem zusammenhängenden Graphen  $|E| \geq |V| - 1$  ist, erhalten wir insgesamt also  $O(|E| \log |E|)$ . Weiters ist ja  $|E| \leq |V|^2$ , also  $\log |E| \leq 2 \log |V|$  und damit  $\log |E| = O(\log |V|)$ . Somit kann die Laufzeit des Algorithmus von Kruskal auch geschrieben werden als  $O(|E| \log |V|)$ .

## 7.2 Der Algorithmus von Prim

Wir betrachten jetzt noch einen zweiten, ebenfalls gierigen, Algorithmus zur Berechnung eines minimalen Spannbaums: den Algorithmus von Prim.

**Definition 7.1.** Sei  $G = (V, E)$  und  $c : E \rightarrow \mathbb{R}_{\geq 0}$ , sei  $G' = (V', E')$  ein Teilgraph von  $G$  und sei  $v \in V \setminus V'$ . Dann sind die *Anschlusskosten* von  $v$  an  $G'$  bezüglich  $c$

$$\min\{c(\{u, v\}) \mid \{u, v\} \in E, u \in V'\}$$

bzw.  $+\infty$  falls diese Menge leer ist.

Der Algorithmus von Prim geht wie folgt vor: gegeben einen Graphen  $G = (V, E)$  mit  $|V| = n$  und eine Kostenfunktion  $c : E \rightarrow \mathbb{R}_{\geq 0}$  setzen wir zu Beginn  $V_1 = \{s\}$ ,  $E_1 = \emptyset$ . Für  $i = 2, \dots, n$  sei  $v_i \in V \setminus V_i$  der Knoten mit minimalen Anschlusskosten an  $(V_i, E_i)$  und  $\{u_i, v_i\}$  eine Kante die diese Anschlusskosten realisiert. Dann setzen wir  $V_{i+1} = V_i \cup \{v_i\}$  und  $E_{i+1} = E_i \cup \{\{u_i, v_i\}\}$ . Der Algorithmus antwortet mit  $(V_n, E_n)$ . Auf diese Weise verwalten wir einen wachsenden Baum der Teilgraph von  $G$  ist.

**Satz 7.3.** *Der Algorithmus von Prim ist korrekt.*

*Beweis.* Zunächst ist leicht zu beobachten dass alle  $B_i = (V_i, E_i)$  Bäume sind. Da  $V_n = V$  ist  $B_n$  ein Spannbaum von  $G$ . Für die Minimalität von  $B_n$  reicht es zu zeigen dass für alle  $i \in \{1, \dots, n\}$  ein minimaler Spannbaum von  $G$  mit Kantenmenge  $M_i$  existiert so dass  $E_i \subseteq M_i$ . Dann ist nämlich  $M_n = E_n$  und damit  $B_n$  minimal. Wir gehen mit Induktion nach  $i$  vor. Der Fall  $i = 1$  ist trivial. Für den Induktionsschritt machen wir eine Fallunterscheidung: Falls  $\{u_i, v_i\} \in M_i$ , sei  $M_{i+1} = M_i$ . Falls  $\{u_i, v_i\} \notin M_i$ , dann muss  $M_i$  eine Kante  $\{w, w'\}$  enthalten

<sup>1</sup>Tatsächlich könnte man das noch genauer abschätzen. Das ist aber für dieses Resultat nicht nötig.

mit  $w \in V_i$  und  $w' \notin V_i$  da  $M_i$  zusammenhängend ist. Dann ist  $\{w, w'\} \neq \{u_i, v_i\}$  und da  $v_i$  minimale Anschlusskosten hat ist  $c(\{u_i, v_i\}) \leq c(\{w, w'\})$ . Wir definieren  $M_{i+1} = (M_i \setminus \{\{w, w'\}\}) \cup \{\{u_i, v_i\}\}$ . Nun ist  $(V_i, E_i)$  und  $G$  zusammenhängend, also gibt es einen Pfad  $p$  von  $w$  nach  $u_i$  und einen Pfad  $q$  von  $v_i$  nach  $w'$ . Somit kann also in jedem  $M_i$ -Pfad die Kante  $\{w, w'\}$  durch  $p, \{u_i, v_i\}, q$  ersetzt werden um daraus einen  $M_{i+1}$ -Pfad zu erhalten.  $M_{i+1}$  ist also zusammenhängend und da  $|M_{i+1}| = |M_i| = |V| - 1$  ist  $M_{i+1}$  mit Satz 2.3 ein Spannbaum von  $G$ . Außerdem ist  $c(M_{i+1}) \leq c(M_i)$  und, da  $M_i$  minimal ist, ist  $c(M_{i+1}) = c(M_i)$  und damit  $(V, M_{i+1})$  ein minimaler Spannbaum von  $G$ .  $\square$

Im Prinzip wäre es hier möglich, den Knoten mit minimalen Anschlusskosten in jeden Schritt neu zu bestimmen. Diese Vorgehensweise kostet aber unnötig Laufzeit. Geschickter ist es zur Implementierung des Algorithmus von Prim die Knoten in einer Minimum-Prioritätswarteschlange abzulegen. Für einen Knoten  $v$  ist  $v.x$  die Priorität, also die minimalen Anschlusskosten bezüglich des aktuellen Baums. Wir führen ein Datenfeld  $B$  mit so dass  $B[v] = \mathbf{wahr}$  ist genau dann wenn der Knoten  $v$  im aktuellen Baum enthalten ist. Die Kanten des Baums werden implizit konstruiert, indem wir mit jedem Knoten  $v$  das Feld  $v.Vorgänger$  mitführen das auf jenen Knoten zeigt mit dem die minimalen Anschlusskosten erreicht werden können. Bei Bedarf kann der Baum dann aus diesen Feldern explizit gemacht werden. Die Laufzeit des Algorithmus von Prim

---

#### Algorithmus 28 Algorithmus von Prim

---

**Prozedur** PRIM( $V, E, c, s$ )

Sei  $B$  ein neues Datenfeld der Länge  $|V|$ , überall mit **falsch** initialisiert

**Für** alle  $v \in V$

$v.x := \infty$

$v.Vorgänger := \text{NIL}$

**Ende Für**

$s.x := 0$

$Q := \text{MIN-PRIORITÄTSWARTESCHLANGE}(V)$

**Solange**  $Q$  nicht leer

$v := \text{EXTRAHIEREMINIMUM}(Q)$

$B[v] := \mathbf{wahr}$

**Für** alle  $(v, w) \in E$

**Falls**  $c(v, w) < w.x$  **und**  $B[w] = \mathbf{falsch}$  **dann**

$w.Vorgänger := v$

REDUZIEREPRIORITÄT( $Q, w, c(v, w)$ )

**Ende Falls**

**Ende Für**

**Ende Solange**

**Antworte**  $(V, \{\{u, v\} \mid u = v.Vorgänger\})$

**Ende Prozedur**

---

setzt sich wie folgt zusammen: Die Initialisierung inklusive Aufbau der Prioritätswarteschlange benötigt Zeit  $O(|V|)$ , ebenso die Berechnung der Antwort. Die äußere Schleife wird  $O(|V|)$  mal durchlaufen wobei die Extraktion des Minimums jeweils  $O(\log |V|)$  Zeit benötigt. Die innere Schleife wird  $|E|$  mal durchlaufen wobei bei jedem Durchlauf durch die Änderung der Priorität Kosten von  $O(\log |V|)$  anfallen können. In einem zusammenhängenden Graphen ist  $|E| = \Omega(|V|)$ , insgesamt erhalten wir also eine Laufzeit von  $O(|E| \log |V|)$ .

## 7.3 Der Algorithmus von Dijkstra

Wir werden jetzt einen eng mit dem Algorithmus von Prim verwandten Algorithmus betrachten: den Algorithmus von Dijkstra. Er löst das Problem der Bestimmung des kürzesten Pfades, vgl. Abschnitt 2.2. Dieses Problem hat viele Anwendungen, z.B. die offensichtliche in Navigationssystemen. Wie beim Algorithmus von Prim werden wir sukzessive einen Teilgraphen  $B$  von  $G$  aufbauen der aus den minimalen Pfaden von  $s$  zu den Knoten von  $B$  besteht. Zur Verwaltung der Knoten verwenden wir eine Minimum-Prioritätswarteschlange. Der wesentliche Unterschied besteht darin, dass die Priorität  $v.x$  eines Knoten  $v$  nicht mehr seine Anschlusskosten sind sondern seine  $B$ -Distanz von  $s$ , d.h. die Länge des kürzesten Pfades von  $s$  nach  $v$  der bis auf die letzte Kante nur aus Knoten in  $B$  besteht. Ebenso wie beim Algorithmus von Prim repräsentieren wir durch  $v.Vorgänger$  implizit alle konstruierten Pfade. Der Algorithmus von Dijkstra berechnet die kürzesten Pfade von  $s$  zu jedem Knoten  $v \in V$ . Die Antwort für ein bestimmtes  $t \in V$  kann daraus leicht in Zeit  $O(|V|)$  bestimmt werden.

---

### Algorithmus 29 Algorithmus von Dijkstra

---

**Prozedur** DIJKSTRA( $V, E, c, s, t$ )

Sei  $B$  ein neues Datenfeld der Länge  $|V|$ , überall mit **falsch** initialisiert

**Für** alle  $v \in V$

$v.x := \infty$

$v.Vorgänger := \text{NIL}$

**Ende Für**

$s.x := 0$

$Q := \text{MIN-PRIORITÄTSWARTESCHLANGE}(V)$

**Solange**  $Q$  nicht leer

$v := \text{EXTRAHIEREMINIMUM}(Q)$

$B[v] := \text{wahr}$

**Für** alle  $(v, w) \in E$

**Falls**  $v.x + c(v, w) < w.x$  **und**  $B[w] = \text{falsch}$  **dann**

$w.Vorgänger := v$

$\text{REDUZIEREPRIORITÄT}(Q, w, v.x + c(v, w))$

**Ende Falls**

**Ende Für**

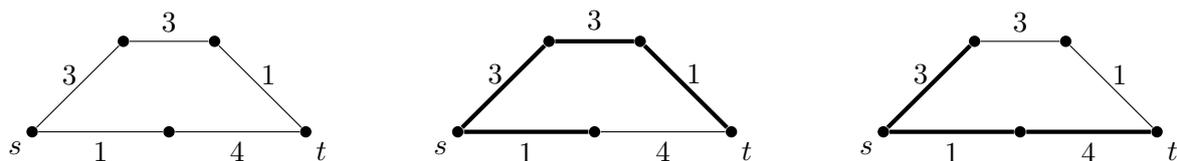
**Ende Solange**

**Antworte**  $(t, t.Vorgänger, t.Vorgänger.Vorgänger, \dots, s)^{-1}$

**Ende Prozedur**

---

*Beispiel 7.3.* Ein Graph  $G$  mit Kantenkosten (links), der vom Algorithmus von Prim bei Eingabe  $G$  erzeugte Spannbaum (Mitte) und die vom Algorithmus von Dijkstra erzeugten Pfade bei Besuch von  $t$  (rechts).



**Satz 7.4.** *Der Algorithmus von Dijkstra ist korrekt.*

*Beweis.* Sei  $G = (V, E)$  sowie  $c : E \rightarrow \mathbb{R}_{\geq 0}$  die Eingabe, sei  $|V| = n$ . Für  $i = 1, \dots, n$  sei  $S_i$  die Menge der nach dem  $i$ -ten Schleifendurchlauf abgearbeiteten, d.h. aus  $Q$  extrahierten, Knoten

und für  $v \in S_i$  sei  $p_v$  der durch die Vorgänger-Felder codierte Pfad von  $s$  nach  $v$ . Wir behaupten dass für alle  $i = 1, \dots, n$  und alle  $v \in S_i$  gilt dass  $p_v$  ein kürzester Pfad von  $s$  nach  $v$  ist. Wir gehen mit Induktion nach  $i$  vor. Für  $i = 1$  ist das trivial. Für den Induktionsschritt sei  $v$  der Knoten in  $S_{i+1} \setminus S_i$  und sei  $u$  der Vorgänger von  $v$ . Dann ist nach Induktionshypothese  $p_u$  ein kürzester Pfad von  $s$  nach  $u$  und  $p_v = p_u, v$ . Sei  $p$  ein anderer Pfad von  $s$  nach  $v$ , dann hat  $p$  die Form  $p_1, x, y, p_2$  wobei  $p_1, x$  in  $S_i$  ist und  $y$  außerhalb von  $S_i$ . Da aber nach Definition des Algorithmus  $v$  unter allen mit einer Kante aus  $S_i$  erreichbaren Knoten in  $V \setminus S_i$  minimale Distanz zu  $s$  hat, ist  $c(p_1, x, y) \geq c(p_v)$  und damit  $c(p) \geq c(p_v)$ .  $\square$

Die Laufzeit vom Algorithmus von Dijkstra ist, wie jene vom Algorithmus von Prim,  $O(|E| \log |V|)$ .

## 7.4 Matroide

**Definition 7.2.** Sei  $E$  eine endliche Menge und  $\mathcal{U} \subseteq \mathfrak{P}(E)$ .  $M = (E, \mathcal{U})$  heißt *Matroid* falls:

1.  $\emptyset \in \mathcal{U}$ ,
2.  $A \subseteq B \in \mathcal{U}$  impliziert  $A \in \mathcal{U}$  und
3. es gilt die folgende *Austauscheigenschaft*:  
Ist  $A, B \in \mathcal{U}$  mit  $|A| > |B|$ , dann gibt es  $x \in A \setminus B$  so dass  $B \cup \{x\} \in \mathcal{U}$ .

Für  $A \in \mathcal{U}$  werden wir auch sagen dass  $A$  eine *unabhängige Menge* ist. Die Bezeichnung Austauschbarkeit rührt daher, dass in der Menge  $A = \{x\} \cup A'$  die Teilmenge  $A'$  durch  $B$  ausgetauscht werden kann ohne die Eigenschaft der Unabhängigkeit zu verlieren. Wegen Bedingung 2. ist Bedingung 1. erfüllt genau dann wenn die Bedingung 1'  $\mathcal{U} \neq \emptyset$  erfüllt ist, was zu einer alternativen Definition führt.

*Beispiel 7.4.* Sei  $M$  eine Matrix über einem Körper,  $E$  die Menge der Spaltenvektoren von  $M$  und  $\mathcal{U}$  die Menge der linear unabhängigen Mengen von Spaltenvektoren von  $M$ . Dann bildet  $(E, \mathcal{U})$  ein Matroid:

Klar ist dass  $\emptyset$  linear unabhängig ist und dass jede Teilmenge einer linear unabhängigen Menge auch linear unabhängig ist,  $(E, \mathcal{U})$  erfüllt also 1. und 2. Für die Austauschbarkeit seien  $A$  und  $B$  linear unabhängig und  $|A| > |B|$ , dann ist  $\dim \langle A \rangle = |A| > |B| = \dim \langle B \rangle$ . Angenommen für alle  $x \in A$  wäre  $B \cup \{x\}$  linear abhängig, dann wäre ja  $\langle A \cup B \rangle = \langle B \rangle$  und damit  $\dim \langle A \cup B \rangle = \dim \langle B \rangle < \dim \langle A \rangle$ . Das widerspricht aber  $\dim \langle A \cup B \rangle \geq \dim \langle A \rangle$ .

Dieser Spezialfall aus der linearen Algebra hat auch historisch den Begriff des Matroids motiviert. Die Austauschbarkeit ist eine Verallgemeinerung des Austauschsatzes von Steinitz. Der Terminologie der linearen Algebra folgend können wir nun definieren und beweisen:

**Definition 7.3.** Sei  $(E, \mathcal{U})$  ein Matroid. Eine Menge  $A \in \mathcal{U}$  heißt *Basis* falls kein  $A'$  existiert mit  $A \subset A' \in \mathcal{U}$ .

Da  $E$  endlich und  $\mathcal{U}$  nicht leer ist hat jedes Matroid mindestens eine Basis.

**Satz 7.5.** Sei  $M = (E, \mathcal{U})$  ein Matroid, seien  $B_1, B_2$  Basen von  $M$ , dann ist  $|B_1| = |B_2|$ .

*Beweis.* Angenommen  $B_1$  und  $B_2$  wären Basen von  $M$  mit  $|B_1| > |B_2|$ , dann gäbe es nach der Austauschbarkeit ein  $x \in B_1 \setminus B_2$  so dass  $B_2 \cup \{x\} \in \mathcal{U}$ , also wäre  $B_2$  nicht maximal unabhängig.  $\square$

**Definition 7.4.** Der *Rang* eines Matroids  $M$  ist die Kardinalität seiner Basen.

*Beispiel 7.5.* Sei  $G = (V, E)$  ein zusammenhängender Graph. Wir bezeichnen  $A \subseteq E$  als unabhängig falls  $(V, A)$  zyklensfrei, d.h. ein Wald, ist. Sei  $\mathcal{U}$  die Menge aller unabhängigen Kantenmengen, dann ist  $M_G = (E, \mathcal{U})$  ein Matroid:

Es ist nämlich  $(V, \emptyset)$  zyklensfrei und falls  $(V, A)$  zyklensfrei ist und  $B \subseteq A$ , dann ist auch  $(V, B)$  zyklensfrei. Für die Austauscheneigenschaft seien  $A, B \in \mathcal{U}$  mit  $|A| > |B|$ . Nach Korollar 2.1 besteht der Wald  $(V, A)$  aus  $|V| - |A|$  Bäumen und der Wald  $(V, B)$  aus  $|V| - |B|$  Bäumen. Nach dem Schubfachprinzip gibt es also in  $(V, A)$  einen Baum  $T$  dessen Knoten zu mindestens zwei Bäumen in  $(V, B)$  gehören. Da  $T$  zusammenhängend ist, gibt es eine Kante  $\{v, w\}$  in  $T$  so dass  $v$  und  $w$  zu verschiedenen Bäumen in  $(V, B)$  gehören. Damit ist  $B \cup \{\{v, w\}\} \in \mathcal{U}$ .

Die Basen von  $M_G$  sind dann die maximalen unabhängigen Mengen, d.h. die maximalen zyklensfreien Teilgraphen, d.h. nach Satz 2.3 die Spann bäume von  $V$ .

**Definition 7.5.** Sei  $(E, \mathcal{M})$  ein Matroid und  $g : E \rightarrow \mathbb{R}_{\geq 0}$ , dann heißt  $(E, \mathcal{M}, g)$  *gewichtetes Matroid*.

Wir können also das Problem der Bestimmung eines minimalen Spannbaums abstrahieren zur Bestimmung einer Basis minimalen Gewichts in einem gewichteten Matroid:

**Basis minimalen Gewichts**

Eingabe: gewichtetes Matroid  $(E, \mathcal{M}, g)$

Ausgabe: Basis  $B \in \mathcal{M}$  so dass  $\sum_{b \in B} g(b)$  minimal

Dann besteht die korrespondierende Abstraktion des Algorithmus von Kruskal in dem generischen gierigen Algorithmus, der in Algorithmus 30 angegeben ist.

---

**Algorithmus 30** Generischer gieriger Algorithmus

---

**Prozedur** GIERIG( $E, \mathcal{U}, g$ )

Sortiere  $E[1, \dots, n]$  so dass  $g(E[1]) \leq \dots \leq g(E[n])$

$A := \emptyset$

**Für**  $i := 1, \dots, n$

**Falls**  $A \cup \{E[i]\} \in \mathcal{U}$  **dann**

$A := A \cup \{E[i]\}$

**Ende Falls**

**Ende Für**

**Antwort**  $A$

**Ende Prozedur**

---

*Beispiel 7.6.* Der generische gierige Algorithmus angewandt auf das in Beispiel 7.5 diskutierte Matroid  $M_G$  entspricht genau dem Algorithmus von Kruskal.

**Satz 7.6.** Sei  $M = (E, \mathcal{U}, g)$  ein gewichtetes Matroid. Dann berechnet GIERIG( $E, \mathcal{U}, g$ ) eine Basis minimalen Gewichts von  $M$ .

*Beweis.* Sei  $(E, \mathcal{U}, g)$  ein gewichtetes Matroid mit Rang  $r$  und seien  $a_1, \dots, a_r \in E$  die vom generischen gierigen Algorithmus ausgewählten Elemente in dieser Reihenfolge. Dann ist  $g(a_1) \leq \dots \leq g(a_r)$ . Sei  $\{b_1, \dots, b_r\}$  eine Basis mit  $g(b_1) \leq \dots \leq g(b_r)$ . Es reicht zu zeigen dass  $g(a_i) \leq g(b_i)$  für alle  $i \in \{1, \dots, r\}$ . Angenommen es gäbe ein  $k \in \{1, \dots, r\}$  so dass  $g(a_k) > g(b_k)$ ,

dann ist  $k \geq 2$  da  $a_1 \in E$  minimales Gewicht hat. Sei  $A = \{a_1, \dots, a_{k-1}\}$  und  $B = \{b_1, \dots, b_k\}$ . Dann ist  $|B| > |A|$  und damit gibt es nach der Austausch Eigenschaft ein  $b_j \in B \setminus A$  so dass  $A \cup \{b_j\} \in \mathcal{U}$ . Dann ist aber  $g(b_j) \leq g(b_k) < g(a_k)$  und damit steht  $b_j$  strikt vor  $a_k$  in der Sortierung  $e_1, \dots, e_n$  von  $E$ . Der gierige Algorithmus hätte also  $b_j$  zu  $A$  hinzugefügt bevor er  $a_k$  betrachtet hätte, Widerspruch.  $\square$

*Beispiel 7.7.* Sei  $G = (V, E)$  ein zusammenhängender Graph,  $g : E \rightarrow \mathbb{R}_{\geq 0}$  und  $s \in V$ . Wir definieren dann  $P$  als die Menge aller bei  $s$  beginnenden zyklensfreien Pfade und erweitern  $g$  auf  $P$  durch  $g(p) = \sum_{e \text{ auf } p} g(e)$ . Eine Menge  $A \subseteq P$  heißt unabhängig wenn die Pfade in  $A$  paarweise verschiedene Endknoten haben. Sei  $\mathcal{U}$  die Menge der unabhängigen Pfadmengen, dann ist  $M = (P, \mathcal{U}, g)$  ein gewichtetes Matroid.

Klar ist  $\emptyset \in \mathcal{U}$  und  $A \subseteq B \in \mathcal{U}$  impliziert  $A \in \mathcal{U}$ . Die Austausch Eigenschaft gilt, da jede unabhängige Menge von  $k + 1$  Pfaden einen Endknoten enthält der nicht Endknoten in einer unabhängigen Menge von  $k$  Pfaden ist.

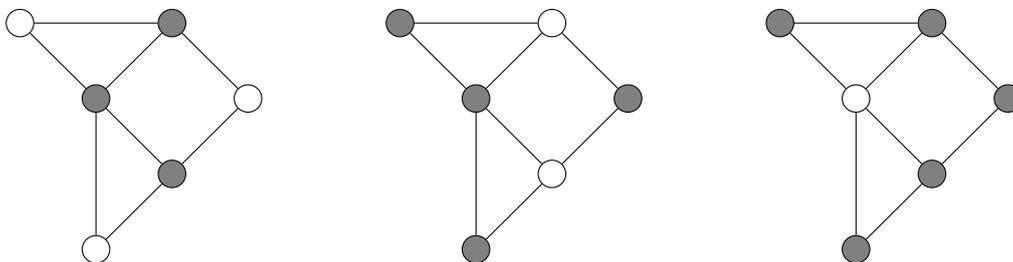
Eine Basis von  $M$  ist eine Menge von Pfaden die alle Knoten erreichen, eine Basis minimalen Gewichts enthält für jeden Knoten  $v$  nur einen kürzesten Pfad von  $s$  nach  $v$ . Der Rang von  $M$  ist  $|V|$ . Der generische gierige Algorithmus entspricht dann in Ablauf und Ergebnis dem Algorithmus von Dijkstra.

## 7.5 Das Knotenüberdeckungsproblem

Die bisher betrachteten gierigen Algorithmen haben jeweils ein Berechnungs- bzw. Optimierungsproblem exakt gelöst, sie bestimmten einen *kürzesten* Pfad, einen *minimalen* Spannbaum, etc. Oft treten in der Praxis aber Probleme auf, für deren exakte optimale Lösung keine effizienten Algorithmen bekannt sind. In einer solchen Situation ist es dann oft nützlich einen *Approximationsalgorithmus* zu verwenden. Ein solcher findet zwar im Allgemeinen keine optimale Lösung, aber er findet eine die, in einem noch zu präzisierenden Sinn, nicht allzu weit davon entfernt liegt. Wesentlich ist dann natürlich dass der Approximationsalgorithmus effizient genug für praktische Anwendungen ist. Für derartige Approximationsverfahren spielen gierige Algorithmen eine wichtige Rolle. Wir werden dazu in diesem Abschnitt ein Beispiel sehen.

**Definition 7.6.** Sei  $G = (V, E)$  ein Graph. Eine *Knotenüberdeckung* von  $V$  ist eine Menge  $V' \subseteq V$  so dass  $\{u, v\} \in E \Rightarrow u \in V'$  oder  $v \in V'$ .

*Beispiel 7.8.* Ein Graph und drei seiner Knotenüberdeckungen (in grau):



Wie man sich leicht überlegen kann hat dieser Graph keine Knotenüberdeckung der Größe 2.

Jeder Graph hat die triviale Knotenüberdeckung  $V = V'$ . Wenn wir nach einer optimalen Knotenüberdeckung fragen, d.h. eine minimaler Kardinalität, dann erhalten wir das folgende Berechnungsproblem:

### Optimale Knotenüberdeckung

Eingabe: ungerichteter Graph  $G = (V, E)$

Ausgabe: Knotenüberdeckung  $V'$  von  $G$  so dass  $|V'|$  minimal ist

Natürlich gibt es oft triviale Approximationsalgorithmen, z.B. im Fall der Knotenüberdeckung einfach  $V$  zu verwenden. Deshalb interessiert man sich im Kontext von Approximationsalgorithmen oft für Approximationsgarantien.

**Definition 7.7.** Wir sagen dass ein Algorithmus für ein Minimierungsproblem *Approximationsrate*  $\rho(n)$  hat falls für jede Eingabe  $x$  der Größe  $n$ , für die vom Algorithmus gelieferte Ausgabe mit Kosten  $c(x)$  und die optimalen Kosten  $c^*(x)$  bei Eingabe  $x$  gilt dass  $\frac{c(x)}{c^*(x)} \leq \rho(n)$ .

Mit Kosten in der obigen Definition ist die Qualität der Lösung gemeint, also z.B. im Fall der Knotenüberdeckung die Anzahl der Knoten in der Überdeckung. Die Frage der Laufzeit eines Algorithmus ist unabhängig von der Frage nach seiner Approximationsrate. Ein Algorithmus mit Approximationsrate  $\rho(n)$  wird oft auch als  $\rho(n)$ -*Approximationsalgorithmus* bezeichnet. Oft ist es möglich, für Optimierungsprobleme, auch wenn kein polynomialer Algorithmus zu ihrer exakten Lösung bekannt ist, einen polynomialen Approximationsalgorithmus zu finden, gelegentlich sogar mit konstanter, d.h. nicht von  $n$  abhängiger, Approximationsrate. Gierige Algorithmen eignen sich dafür besonders gut, da sie immer eine geringe Laufzeit haben. Betrachten wir zum Beispiel Algorithmus 31 zur Berechnung einer Knotenüberdeckung. Dieser berechnet keine opti-

---

#### Algorithmus 31 Gierige Knotenüberdeckung

---

**Prozedur** GIERIGEKNOTENÜBERDECKUNG( $V, E$ )

$U := \emptyset$

$A := E$

**Solange**  $A$  nicht leer

Sei  $\{u, v\} \in A$

$U := U \cup \{u, v\}$

Entferne alle  $e$  aus  $A$  die  $u$  oder  $v$  enthalten

**Ende Solange**

**Antworte**  $U$

**Ende Prozedur**

---

male Knotenüberdeckung. Das sieht man allein schon daran, dass er nur Knotenüberdeckungen gerader Kardinalität berechnet, es gibt aber Graphen deren optimale Knotenüberdeckung ungerade Kardinalität hat, siehe Beispiel 7.8. Dieser Algorithmus hat Laufzeit  $O(|E|)$ .

**Satz 7.7.** Die Prozedur GIERIGEKNOTENÜBERDECKUNG ist ein 2-Approximationsalgorithmus.

*Beweis.* Sei  $G = (V, E)$  der Eingabegraph, sei  $V^* \subseteq V$  eine optimale Knotenüberdeckung, sei  $V'$  die Ausgabe des Algorithmus und sei  $E'$  die Menge der aus  $A$  gewählten Kanten, dann ist  $|V'| = 2|E'|$ . Nun gibt es nach der Definition des Algorithmus keine zwei Kanten in  $E'$  die einen Knoten teilen. Also muss eine beliebige Knotenüberdeckung für jede Kante  $e \in E'$  mindestens einen Knoten enthalten, um  $e$  zu überdecken. Insbesondere gilt das auch für  $V^*$ . Damit ist  $|V^*| \geq |E'| = \frac{1}{2}|V'|$  und damit  $\frac{|V'|}{|V^*|} \leq 2$ .  $\square$

Es ist nicht bekannt ob das Problem der optimalen Knotenüberdeckung durch einen Algorithmus mit polynomialer Laufzeit (exakt) gelöst werden kann. An sich kann das schon Grund genug

sein, sich einen Approximationsalgorithmus zu überlegen. In diesem (und vielen vergleichbaren Fällen) weiß man allerdings sogar dass die Existenz eines solchen Algorithmus implizieren würde dass  $\mathbf{P} = \mathbf{NP}$ . Die Frage ob  $\mathbf{P} = \mathbf{NP}$  ist eines der schwierigsten offenen Probleme der Mathematik.

Um das genauer zu erklären, muss kurz ausgeholt werden. Ein Entscheidungsproblem ist ein Berechnungsproblem dessen Ausgabe entweder “ja” oder “nein” ist. Berechnungsprobleme und insbesondere auch Optimierungsprobleme können üblicherweise recht direkt in Entscheidungsprobleme übertragen werden, z.B.

<p><b>Knotenüberdeckung</b></p> <p>Eingabe: endlicher Graph <math>G = (V, E)</math>, <math>k \in \mathbb{N}</math></p> <p>Ausgabe: “ja” wenn <math>G</math> eine Knotenüberdeckung <math>V'</math> hat mit <math> V'  \leq k</math> und “nein” sonst</p>
--

Ein Entscheidungsproblem wird als Menge betrachtet indem es mit der Menge seiner “ja”-Instanzen identifiziert wird. So wird zum Beispiel das Entscheidungsproblem der Knotenüberdeckung identifiziert mit der Menge

$$K = \{(G, k) \mid G \text{ ist endlicher Graph der Knotenüberdeckung } V' \text{ mit } |V'| \leq k \text{ hat}\}.$$

Wir schränken unsere Betrachtung nun ein auf Entscheidungsprobleme die Teilmengen von  $\{0, 1\}^*$  sind. Graphen, natürliche Zahlen, Tupel von Graphen und natürlichen Zahlen, etc. können als Elemente eines Entscheidungsproblems betrachtet werden indem eine Kodierung in Wörter über  $\{0, 1\}$  fixiert wird und der Graph, die Zahl, etc. mit seiner/ihrer Codierung identifiziert wird.

$\mathbf{P}$  bezeichnet nun die Menge aller Entscheidungsprobleme für die ein Algorithmus existiert dessen Laufzeit polynomial in der Größe der Eingabe ist<sup>2</sup>. Ein Entscheidungsproblem  $S$  ist<sup>3</sup> in  $\mathbf{NP}$  genau dann wenn es ein Entscheidungsproblem  $R \in \mathbf{P}$  sowie ein  $c \in \mathbb{N}$  gibt so dass

$$x \in S \Leftrightarrow \exists y \in \{0, 1\}^* \text{ mit } |y| \leq |x|^c \text{ und } (x, y) \in R.$$

An der Definition des Entscheidungsproblems  $K$  der Knotenüberdeckung ist jetzt unmittelbar sichtbar, dass es in  $\mathbf{NP}$  ist indem wir für  $R$  die folgende Relation wählen:

$$((G, k), V') \in R \Leftrightarrow V' \text{ ist Knotenüberdeckung von } G \text{ mit } |V'| \leq k$$

und beobachten dass  $R \in \mathbf{P}$ . Dann gibt es ein  $c$  so dass

$$((V, E), k) \in K \Leftrightarrow \exists y \in \{0, 1\}^* \text{ mit } |y| \leq |x|^c, x \text{ codiert } ((V, E), k), \\ y \text{ codiert } V' \subseteq V \text{ und } ((V, E), V') \in R$$

Überdies kann vom Entscheidungsproblem der Knotenüberdeckung (mit etwas mehr technischem Aufwand) gezeigt werden, dass es  $\mathbf{NP}$ -vollständig ist.  $\mathbf{NP}$ -vollständige Probleme sind

<sup>2</sup>Diese Definition bleibt insofern informell als wir im Rahmen dieser Vorlesung nicht formell definieren was ein Algorithmus ist. Für die Definition von  $\mathbf{P}$  hat das aber kaum Bedeutung da sich unterschiedliche derartige formale Definitionen modulo polynomialer Berechenbarkeit nicht unterscheiden.

<sup>3</sup> $\mathbf{NP}$  steht für “lösbar in nicht-deterministisch polynomialer Zeit” nach einer anderen Definition von  $\mathbf{NP}$  über nicht-deterministische Berechnung.

in einem gewissen technischen Sinn die schwierigsten Probleme in **NP**. Für ein beliebiges **NP**-vollständiges Problem  $S$  gilt dass  $S \in \mathbf{P} \Leftrightarrow \mathbf{P} = \mathbf{NP}$ .

Wenn nun das Optimierungsproblem der optimalen Knotenüberdeckung einen polynomialen Algorithmus besitzt, dann hat auch das Entscheidungsproblem einen polynomialen Algorithmus und, da dieses **NP**-vollständig ist, wäre dann  $\mathbf{P} = \mathbf{NP}$ .

Es gibt tausende **NP**-vollständige Probleme von denen viele praxisrelevant sind. Die Bedeutung der Frage ob  $\mathbf{P} = \mathbf{NP}$  rührt daher dass die (meisten) Probleme in  $\mathbf{P}$  in der Praxis effizient lösbar sind und viele praxisrelevante Probleme **NP**-vollständig sind.

## Kapitel 8

# Dynamische Programmierung

In Kapitel 3 haben wir Teile-und-herrsche Algorithmen kennengelernt. Diese gehen so vor, dass sie eine Instanz eines Problems in kleinere, disjunkte, Teilinstanzen zerlegen, diese unabhängig voneinander lösen und deren Lösungen dann zu einer der ursprünglichen Instanz kombinieren. Solche Algorithmen sind nicht effizient, wenn im Laufe der Berechnung die selben Teilinstanzen oft auftreten. Dann werden diese nämlich unabhängig voneinander mehrfach gelöst. *Dynamische Programmierung* ist ein Designprinzip für Algorithmen, das solche mehrfach auftretenden Teilinstanzen auf eine Weise organisiert die doppelte Berechnungen vermeidet. Damit das effizient möglich ist, ist es notwendig dass die Anzahl der zur Bestimmung einer Lösung notwendigen Teilinstanzen nicht allzu groß ist (also z.B. nur polynomial).

Dieses Prinzip kann gut am folgenden Berechnungsproblem illustriert werden:

### Anzahl der Pfade

Eingabe: endlicher gerichteter azyklischer Graph  $(V, E)$ ,  $s, t \in V$

Ausgabe: Anzahl der Pfade von  $s$  nach  $t$

Zunächst beobachten wir dass die Anzahl  $n_{v,t}$  der Pfade von  $v$  nach  $t$  gleich  $\sum_{(v,w) \in E} n_{w,t}$  ist. Eine Prozedur die diese Beobachtung direkt der teile-und-herrsche Methode folgend realisiert ist in Algorithmus 32 angegeben. Dieser Algorithmus ist zwar korrekt aber nicht sehr effizient.

---

#### Algorithmus 32 Anzahl der Pfade (exponentiell)

---

**Prozedur** ANZAHLPFADE( $E, v, t$ )

**Falls**  $v = t$  **dann**

$n := 1$

**sonst**

$n := 0$

**Für**  $(v, w) \in E$

$n := n + \text{ANZAHLPFADE}(E, w, t)$

**Ende Für**

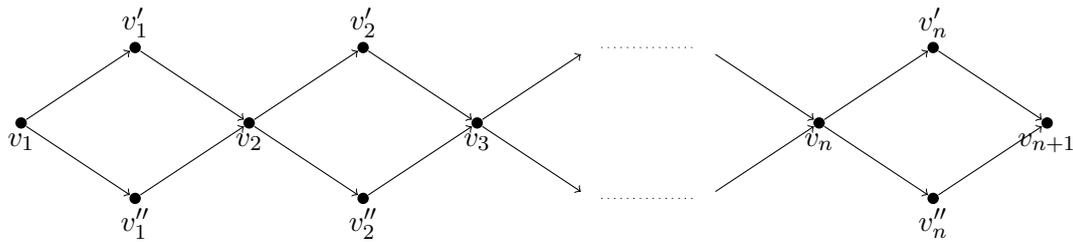
**Ende Falls**

**Antworte**  $n$

**Ende Prozedur**

---

Zum Beispiel wird im Graphen



bei der Berechnung von  $\text{ANZAHLPFAD}(E, v_1, v_{n+1})$  die Prozedur zwei Mal auf  $v_2$  aufgerufen, vier Mal auf  $v_3, \dots, 2^{n-1}$  mal auf  $v_n$ , insgesamt benötigt sie also Laufzeit  $\Omega(2^n)$  auf einem Graphen mit  $|V| = O(n)$  und  $|E| = O(n)$ . Man sieht also dass dieser Algorithmus nicht nur die Anzahl der Pfade berechnet, sondern dass er sie auch alle durchläuft und dadurch im schlechtesten Fall exponentielle Laufzeit benötigt.

Wir interessieren uns aber nicht für die Menge der Pfade, sondern nur für deren Kardinalität. So können wir die zur Entwicklung eines effizienteren Algorithmus entscheidene Beobachtung machen, dass es ausreicht pro  $v \in V$  nur einmal zu berechnen wie viele Pfade von  $v$  nach  $t$  führen, selbst wenn  $v$  von  $s$  aus auf verschiedenen Pfaden erreichbar ist. Eine, effizientere, Vorgehensweise kann nun auf zwei Arten realisiert werden: 1. von oben nach unten wobei bereits berechnete Werte zwischengespeichert werden (*top-down mit Memoisierung*) oder 2. von unten nach oben (*bottom up*), dann entfällt die Notwendigkeit der expliziten Zwischenspeicherung. Die erste ist in Algorithmus 33 angegeben, ein Beispiel für die zweite sehen wir im nächsten Abschnitt. Algorithmus 33 erweitert die Datenstruktur für einen Knoten  $v$  um ein Attribut  $n$

---

**Algorithmus 33** Anzahl der Pfade (top-down mit Memoisierung)

---

**Prozedur**  $\text{ANZAHLPFAD}(E, v, t)$

Setze  $v.n := +\infty$  für alle  $v \in V$

**Antworte**  $\text{ANZAHLPFADEREK}(E, s, t)$

**Ende Prozedur**

**Prozedur**  $\text{ANZAHLPFADEREK}(E, v, t)$

**Falls**  $v.n = +\infty$  **dann**

**Falls**  $v = t$  **dann**

$v.n := 1$

**sonst**

$v.n := 0$

**Für**  $(v, w) \in E$

$v.n := v.n + \text{ANZAHLPFADEREK}(E, w, t)$

**Ende Für**

**Ende Falls**

**Ende Falls**

**Antworte**  $v.n$

**Ende Prozedur**

---

das die Anzahl der Pfade von  $v$  nach  $t$  angibt. Der Wert  $+\infty$  bedeutet dabei dass diese Anzahl noch nicht bekannt ist. Dieser Algorithmus hat die Laufzeit  $O(|V| + |E|)$ . Man beachte auch die Ähnlichkeit dieses Verfahrens mit der Tiefensuche.

## 8.1 Das Stabzerlegungsproblem

Wir betrachten nun das folgende Optimierungsproblem: gegeben ist ein Eisenstab der Länge  $n \in \mathbb{N}$  sowie eine Liste von Preisen  $p_1, \dots, p_n$  wobei  $p_i$  der erzielbare Preis für einen Eisenstab der Länge  $i$  ist. Der gegebene Stab der Länge  $n$  soll so zerschnitten werden, dass der durch Verkauf der Einzelteile erzielbare Gesamtpreis (d.h. die Summe der Preise der Einzelteile) maximal ist.

### Stabzerlegungsproblem

Eingabe:  $n \in \mathbb{N}, p_1, \dots, p_n \in \mathbb{N}$

Ausgabe:  $i_1, \dots, i_k \in \mathbb{N}$  so dass  $n = i_1 + \dots + i_k$  und  $p_{i_1} + \dots + p_{i_k}$  maximal

Für einen Stab der Länge  $n$  gibt es  $2^{n-1}$  verschiedene Möglichkeiten eine Zerteilung zu wählen, da an jedem ganzzahligen Punkt entweder geschnitten oder nicht geschnitten werden kann. Alle diese Zerteilungen durchzuprobieren und davon jene mit maximalem Gesamtpreis zu wählen würde also Zeit  $\Omega(2^n)$  benötigen. Mit dynamischer Programmierung ist das auf effizientere Weise lösbar.

*Beispiel 8.1.* Gegeben  $n = 8$  und  $p_1, \dots, p_n$  wie folgt:

$i$	1	2	3	4	5	6	7	8
$p_i$	1	3	6	9	13	15	16	18

Dann ist die optimale Zerlegung des Stabs der Länge 8 in einen Stab der Länge 3 und einen der Länge 5. Damit wird ein Gesamtpreis von  $p_3 + p_5 = 19$  erzielt.

Sei, für  $1 \leq i \leq n$ ,  $g_i$  der für einen Stab der Länge  $i$  mit Preisen  $p_1, \dots, p_n$  maximal erzielbare Gewinn. Dann ist

$$g_i = \max\{p_1 + g_{i-1}, p_2 + g_{i-2}, \dots, p_{i-1} + g_1, p_i\}$$

da wir unterscheiden können ob nicht geschnitten wird ( $p_i$ ) oder mindestens einmal geschnitten wird und dann nach Länge des ersten Stücks. Wir sehen also dass die optimale Lösung (die  $g_i$  bestimmt) zusammengesetzt ist durch optimale Lösungen für kleinere Instanzen des selben Problems (jene die die  $g_j$  bestimmen für  $1 \leq j < i$ ). Das ist entscheidend für die Anwendbarkeit des dynamischen Programmierens. Außerdem ist die Gesamtanzahl der kleineren Instanzen, d.h. der  $g_j$ , die rekursiv zur Bestimmung von  $g_i$  benötigt werden und damit die Anzahl der Knoten im Teilproblemgraph klein (hier gleich  $i$ ). Wenn wir  $g_0 = 0$  setzen können wir  $g_i$  auch schreiben als

$$g_i = \max\{p_j + g_{i-j} \mid 1 \leq j \leq i\}.$$

Wir können jetzt eine bottom-up Berechnung der  $g_i$  in einem Datenfeld  $G$  durchführen, d.h. beginnend bei kleinen  $i$  in Richtung der größeren  $i$ . Um nicht nur den maximal erzielbaren Gewinn  $g_n$  sondern auch die Teilung  $i_1 + \dots + i_k = n$  des Stabs zu berechnen, verwenden wir das zusätzliche Datenfeld  $S$  das, für  $i = 1, \dots, n$ , in  $S[i]$  speichert wie lange das erste Stück der optimalen Zerteilung eines Stabs der Länge  $i$  ist. Die globale Zerteilung kann dann leicht aus  $S$  in Zeit  $O(n)$  berechnet werden. Diese Vorgehensweise ist in Algorithmus 34 als Pseudocode formuliert. Die Laufzeit dieser Prozedur ist aufgrund der beiden verschachtelten Schleifen  $\Theta(n^2)$ .

---

**Algorithmus 34** Stabzerlegung (bottom-up)

---

**Prozedur** STABZERLEGUNG( $P, n$ )Sei  $G[0, \dots, n]$  neues DatenfeldSei  $S[1, \dots, n]$  neues Datenfeld $G[0] := 0$ **Für**  $i = 1, \dots, n$  $m := 0$ **Für**  $j = 1, \dots, i$ **Falls**  $P[j] + G[i - j] \geq m$  **dann** $m := P[j] + G[i - j]$  $S[i] := j$ **Ende Falls****Ende Für** $G[i] := m$ **Ende Für****Antworte** ( $G[n], S$ )**Ende Prozedur**

---

## 8.2 Segmentierte Methode der kleinsten Quadrate

Bei der einfachen linearen Regression in der Statistik sind Punkte  $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n) \in \mathbb{R}^2$  gegeben die z.B. aus einem Experiment stammen und wir wollen die (Ausgabe-)werte  $y_i$  durch die (Eingabe-)werte  $x_i$  erklären wobei wir annehmen, dass ein linearer Zusammenhang besteht. Wir wollen also eine Gerade  $y = ax + b$  bestimmen welche die Punkte  $p_1, \dots, p_n$  bestmöglich im Sinne der kleinsten Fehlerquadrate annähert, d.h. es sollen  $a, b \in \mathbb{R}$  bestimmt werden so dass die *Fehlerquadratsumme*

$$\sum_{i=1}^n (y_i - ax_i - b)^2$$

minimiert wird. Man kann zeigen dass diese Lösung eindeutig ist und gegeben ist durch

$$a = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad \text{und} \quad b = \bar{y} - a\bar{x}.$$

wobei  $\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$  und  $\bar{y} = \frac{\sum_{i=1}^n y_i}{n}$ .

Oft können die gegebenen Daten aber durch eine einzelne Gerade nicht sehr gut angenähert werden, siehe Abbildung 8.1. Eine Reaktion darauf, die wir hier verfolgen wollen, besteht darin, die Daten stückweise durch Geraden anzunähern. Dabei muss nun natürlich spezifiziert werden was minimiert werden soll, d.h. wie die Anzahl der Geraden gegen die Fehlerquadratsumme gewichtet werden soll. Eine Möglichkeit dies zu tun besteht darin, eine Konstante  $A > 0$  festzulegen, welche die Kosten der Verwendung einer (zusätzlichen) Gerade darstellt. Dann werden die Punkte  $p_1, \dots, p_n$  entlang der  $x$ -Achse in Segmente geteilt von denen jedes unabhängig durch eine Gerade approximiert wird. Je nach Wert von  $A$  fällt die Anzahl der Segmente mehr oder weniger stark ins Gewicht. Wir kommen so zum folgenden Optimierungsproblem

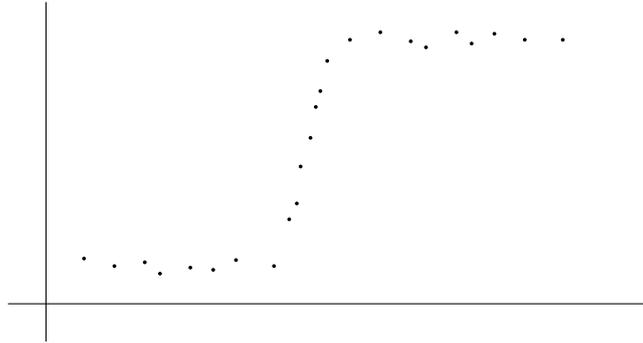


Abbildung 8.1: Eingabedaten für segmentierte lineare Regression

**Segmentierte einfache lineare Regression**

Eingabe:  $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n) \in \mathbb{R}^2$  mit  $x_1 < \dots < x_n$  und  $A > 0$

Ausgabe:  $0 = n_0 < n_1 < \dots < n_{k-1} < n_k = n$  und für alle  $i = 1, \dots, k$  eine Gerade  $y = a_i x + b_i$  so dass

$$\sum_{i=1}^k \left( A + \sum_{j=n_{i-1}+1}^{n_i} (y_j - a_i x_j - b_i)^2 \right)$$

minimal ist

Auch dieses Problem eignet sich für einen Ansatz mit dynamischem Programmieren. Für  $j = 1, \dots, n$  seien  $c_j$  die Kosten der optimalen Lösung des Teilproblems  $p_1, \dots, p_j$ . Dann setzt sich  $c_j$  zusammen aus den Kosten des letzten Segments  $p_i, \dots, p_j$  sowie den Kosten der optimalen Lösung des Teilproblems  $p_1, \dots, p_{i-1}$  und den Kosten der Verwendung einer Geraden. Wir erhalten also

$$c_j = \min\{e_{i,j} + A + c_{i-1} \mid 1 \leq i \leq j\}$$

wobei  $e_{i,j}$  die minimale Fehlerquadratsumme des Segments  $p_i, \dots, p_j$  ist. Außerdem sei  $c_0 = 0$ , womit der Fall dass  $p_1, \dots, p_j$  nicht mehr unterteilt wird abgedeckt wird durch  $c_j = e_{1,j} + A + c_0 = e_{1,j} + A$ . Ähnlich wie bei der Stabzerlegung bildet diese Beobachtung nun die Basis für einen Algorithmus der die mehrfache Betrachtung von Teilproblemen vermeidet, siehe Algorithmus 35. Für ein Segment von  $k$  Punkten kann die Gerade mit minimaler Fehlerquadratsumme in Zeit  $O(k)$  über die oben angegebenen Formeln bestimmt werden. Damit kann die kleinste Fehlerquadratsumme dieser Gerade in Zeit  $O(k)$  bestimmt werden. Auf diese Weise alle  $E[i, j]$  mit  $i \leq j \leq n$  zu berechnen benötigt also Zeit  $O(n^3)$ . Die beiden verschachtelten Schleifen im zweiten Teil des Algorithmus benötigen Zeit  $O(n^2)$ . Wir erhalten also insgesamt eine Laufzeit von  $O(n^3)$ . Die Laufzeit dieses Algorithmus kann durch eine geschicktere Behandlung der ersten Phase auf  $O(n^2)$  reduziert werden. Da dies aber nicht mehr viel zur Diskussion der dynamischen Programmierung beiträgt wollen wir hier darauf verzichten.

---

**Algorithmus 35** Segmentierte Methode der kleinsten Quadrate

---

**Prozedur** SEGMENTIERTEMKQ( $P, n, A$ )

Sei  $E[1, \dots, n; 1, \dots, n]$  ein neues Datenfeld

**Für** alle  $1 \leq i \leq j \leq n$

$E[i, j] :=$  minimale Fehlerquadratsumme für das Segment  $p_i, \dots, p_j$

**Ende Für**

Sei  $C[0, \dots, n]$  ein neues Datenfeld

Sei  $S[1, \dots, n]$  ein neues Datenfeld

$C[0] := 0$

**Für**  $j = 1, \dots, n$

$m := +\infty$

**Für**  $i = 1, \dots, j$

**Falls**  $E[i, j] + A + C[i - 1] < m$  **dann**

$m := E[i, j] + A + C[i - 1]$

$S[j] := i$

**Ende Falls**

**Ende Für**

$C[j] := m$

**Ende Für**

**Antworte** ( $C[n], S$ )

**Ende Prozedur**

---

# Kapitel 9

## Randomisierung

Unter einem *randomisierten Algorithmus* versteht man einen Algorithmus der gewisse Entscheidung basierend auf (aus externer Quelle erhaltenen) Zufallszahlen macht<sup>1</sup>. Das kann zu verschiedenen Zwecken nützlich sein. Einerseits kann man die Eingabedaten oder das Verhalten des Algorithmus randomisieren und so sicherstellen, dass ein Algorithmus auch im schlechtesten Fall nur die Komplexität des durchschnittlichen Falls hat. Dafür hat der Algorithmus dann auch im besten Fall die Komplexität des durchschnittlichen Falls. Je nach Laufzeit im schlechtesten, besten und durchschnittlichen Fall und je nach Verteilung der Eingabedaten ist das mehr oder weniger nützlich. Andererseits kann Randomisierung in einem Algorithmus auch verwendet werden, um schnell ein Resultat zu liefern, das aber nur mit gewisser Wahrscheinlichkeit  $< 1$  korrekt ist. Eine Wiederholung (mit unabhängigen Zufallszahlen) erlaubt dann eine beliebige Annäherung der Irrtumswahrscheinlichkeit an 0.

### 9.1 Randomisierung der Eingabe

In Hinblick auf ein einfaches Beispiel für die Randomisierung der Eingabe wollen wir das folgende *Bewerberproblem* betrachten. Eine Abfolge von Bewerbern absolviert Vorstellungsgespräche für eine Stelle. Einer hire&fire-Mentalität folgend soll zu jedem Zeitpunkt, d.h. auch zwischen den Bewerbungsgesprächen, der beste Bewerber die Stelle halten. Wenn ein besserer Bewerber als der aktuelle Stelleninhaber gefunden wird, wird dieser durch jenen ersetzt. Wir gehen davon aus dass die Bewerber mit  $k_1, \dots, k_n \in \mathbb{R}_{>0}$  beurteilt werden. Die natürliche Vorgehensweise ist in Algorithmus 36 als Pseudocode ausformuliert. Wir gehen dabei davon aus dass  $K[i]$  den Wert  $k_i$  enthält und dass  $K[0] = 0$  ist. Für die Analyse wollen wir davon ausgehen, dass die Einstellung eines Kandidaten hohe Kosten verursacht. Wir interessieren uns also dafür, wie oft Zeile 5 in dieser Prozedur ausgeführt wird. Eine Vorgehensweise wie in Algorithmus 36 zur Suche eines Maximums oder Minimums kommt oft als Teil anderer Algorithmen vor.

**Lemma 9.1.** *Die Anzahl der Neuanstellungen ist im schlechtesten Fall  $n$  und im durchschnittlichen Fall  $O(\log n)$ .*

*Beweis.* Der schlechteste Fall tritt ein wenn  $k_1 < k_2 < \dots < k_n$ . Dann werden  $n$  Neuanstellungen durchgeführt, eine für jeden Kandidaten.

---

<sup>1</sup>Das ist nicht zu verwechseln mit einem nicht-deterministischen Algorithmus der, in gewissem Sinn, verschiedene deterministische Berechnungen gleichzeitig macht.

---

**Algorithmus 36** Direkte Lösung des Bewerberproblems

---

```
1: Prozedur BEWERBERSUCHE( $K, n$ )
2:    $m := 0$ 
3:   Für  $i := 1, \dots, n$ 
4:     Falls  $K[i] > K[m]$  dann                                 $\triangleright$  Vorstellungsgespräch für Kandidat  $i$ 
5:        $m := i$                                                $\triangleright$  Einstellen von Kandidat  $i$ 
6:     Ende Falls
7:   Ende Für
8:   Antworte  $m$ 
9: Ende Prozedur
```

---

Für den durchschnittlichen Fall gehen wir davon aus, dass jede Eingabepermutation der  $k_i$  gleich wahrscheinlich ist. Sei  $X$  die Anzahl der Neuanstellungen und sei

$$X_i = \begin{cases} 1 & \text{falls der } i\text{-te Kandidat eingestellt wird und} \\ 0 & \text{falls der } i\text{-te Kandidat nicht eingestellt wird.} \end{cases}$$

Dann ist  $X = \sum_{i=1}^n X_i$  und, für  $i = 1, \dots, n$  ist  $EX_i = W(X_i = 1)$ . Da der  $i$ -te Kandidat eingestellt wird genau dann wenn  $k_i = \max\{k_1, \dots, k_i\}$ , ist nach Lemma 2.1 die Wahrscheinlichkeit  $W(X_i = 1) = \frac{1}{i}$ . Somit erhalten wir

$$EX = E \sum_{i=1}^n X_i = \sum_{i=1}^n EX_i = \sum_{i=1}^n \frac{1}{i} = H_n = \ln n + O(1).$$

□

Wir haben also einen Algorithmus der im schlechtesten Fall deutlich höhere Kosten verursacht als im durchschnittlichen Fall. Um die Laufzeit des durchschnittlichen Falls zu garantieren, können wir die Eingabedaten vorverarbeiten indem wir auf sie eine (uniform verteilte) Zufallspermutation anwenden. Sei ZUFALLSPERMUTATION( $n$ ) eine solche uniform verteilte Zufallspermutation von  $n$  Elementen. Wir erhalten dann den in Algorithmus 37 angegebenen Pseudocode. Der Erwartungswert der Anzahl der Neuanstellungen wird hier über die Verteilung der Eingabe

---

**Algorithmus 37** Randomisierte Lösung des Bewerberproblems

---

```
Prozedur BEWERBERSUCHERANDOMISIERT( $K, n$ )
   $K :=$  ZUFALLSPERMUTATION( $K$ )
  Antworte BEWERBERSUCHE( $K, n$ )
Ende Prozedur
```

---

und der Zufallspermutation gebildet und ist damit unabhängig von der Eingabe  $O(\log n)$ . Die systematische Konstruktion einer Eingabe mit schlechtem Verhalten ist also nicht mehr möglich. In diesem Sinn gibt es keine worst-case Eingabe mehr. Der Nachteil von Algorithmus 37 ist, dass es in diesem Sinn auch keine best-case Eingabe mehr gibt. Wenn wir also erwarten sehr häufig eine annähernd absteigend sortierte Eingabe zu erhalten, wird Algorithmus 36 vorzuziehen sein. Eine vergleichbare Situation trat auch in Abschnitt 5.2 auf. Das auf (nicht-balancierten) Suchbäumen basierende Sortierverfahren hatte eine Laufzeit von  $O(n^2)$  im schlechtesten Fall aber  $O(n \log n)$  im Durchschnittsfall. Durch eine Zufallspermutation der Eingabe kann dieses in ein randomisiertes Verfahren umgewandelt werden dass auf beliebiger Eingabe eine erwartete Laufzeit von  $O(n \log n)$  hat.

Wir wollen uns jetzt damit beschäftigen wie man eine Zufallspermutation erzeugen kann. Klar ist, dass wir irgendeine Art von Zufallsquelle voraussetzen müssen. Wir nehmen also die Verfügbarkeit einer Prozedur  $ZUFALL(i, j)$  an, die eine uniform verteilte Zufallszahl im Intervall  $[i, j] \subseteq \mathbb{N}$  liefert. Viele Programmiersprachen bzw. Betriebssysteme stellen tatsächlich solche Funktionen zur Verfügung, die als Zufallsquelle verschiedene Umgebungsdaten oder einen Pseudozufallszahlen-Generator verwenden.

Ein geeigneter Ansatz zur Berechnung einer zufälligen Permutation eines Datenfelds  $A$  der Länge  $n$  besteht darin, für  $i = 1, \dots, n$  das Element  $A[i]$  mit einem zufälligen Element in  $A[i, \dots, n]$  zu vertauschen. Das ist der Algorithmus von Fisher-Yates, als Pseudocode ausformuliert in Algorithmus 38. Dieser Algorithmus hat Laufzeit  $\Theta(n)$  (unter der Annahme dass  $ZUFALL(i, j)$

---

**Algorithmus 38** Der Fisher-Yates Algorithmus

---

**Prozedur** FISHERYATES( $A$ )

**Für**  $i = 1, \dots, A.Länge$

Vertausche  $A[i]$  und  $A[ZUFALL(i, n)]$

**Ende Für**

**Antworte**  $A$

**Ende Prozedur**

---

konstante Zeit benötigt). Klar ist, dass der Algorithmus von Fisher-Yates eine Permutation des Eingabdatenfelds berechnet, da er nur Vertauschungen durchführt. Für die Analyse der Wahrscheinlichkeitsverteilung der erzeugten Permutation ist noch etwas Arbeit nötig.

**Satz 9.1.** *Der Fisher-Yates Algorithmus erzeugt eine uniform verteilte Zufallspermutation.*

*Beweis.* Sei  $n = A.Länge$  und für  $i = 0, 1, \dots, n$  sei  $A_i$  das Datenfeld nach dem  $i$ -ten Durchlauf der Schleife. Dann ist  $A_0$  das Eingabdatenfeld. Sei o.B.d.A.  $A_0[i] = i$  für  $i = 1, \dots, n$ . Sei  $V_k^n$  die Menge aller Variationen von  $k$  aus  $n$  Elementen ohne Wiederholung. Dann ist  $|V_k^n| = \frac{n!}{(n-k)!}$ . Wir zeigen mit Induktion nach  $i = 1, \dots, n$  dass für alle  $v \in V_i^n$ :  $W(A_i[1, \dots, i] = v) = \frac{1}{|V_i^n|} = \frac{(n-i)!}{n!}$ . Für  $i = n$  folgt daraus dass das Ausgabdatenfeld  $A_n$  eine uniform verteilte Permutation ist. Die Induktionsbasis  $i = 1$  folgt direkt aus Definition des Algorithmus. Für den Induktionsschritt sei  $w \in V_{i+1}^n$ , dann ist  $w = \langle v, x \rangle$  wobei  $v \in V_i^n$  und  $x \in \{1, \dots, n\} \setminus v$ . Dann ist

$$W(A_{i+1}[1, \dots, i+1] = w) = W(A_i[1, \dots, i] = v \text{ und } A_{i+1}[i+1] = x)$$

und da  $\{1, \dots, n\} \setminus v$  genau die Elemente von  $A_i[i+1, \dots, n]$  sind ist  $W(A_{i+1}[i+1] = x) = \frac{1}{n-i}$

$$= \frac{(n-i)!}{n!} \frac{1}{n-i} = \frac{(n-(i+1))!}{n!}.$$

□

## 9.2 Quicksort

Quicksort ist einer der schnellsten Sortieralgorithmen und wird deshalb in der Praxis oft verwendet. Der Algorithmus folgt dem Teile-und-Herrsche Prinzip. Das Eingabdatenfeld  $A[1, \dots, n]$  wird zunächst durch Vertauschungen von Elementen in zwei Teile geteilt:  $A[1, \dots, m-1]$  und  $A[m+1, \dots, n]$  so dass alle Elemente im linken Teil kleiner gleich  $A[m]$  sind und alle Elemente im rechten Teil größer gleich  $A[m]$  sind. Dann wird die Prozedur rekursiv auf den beiden Teilen aufgerufen. Insgesamt wird so das Datenfeld durch Vertauschungen sortiert.

Die Aufteilung des Datenfelds  $A[l, \dots, r]$  wird so realisiert, dass zunächst ein *Pivotelement* aus dem Datenfeld gewählt wird, z.B.  $A[r]$ . Danach wird das Datenfeld von links nach rechts durchlaufen wobei der jeweils bisher durchlaufene Teil aus zwei Teilen besteht: jene Elemente die größer sind als  $A[r]$  und jene die kleiner sind als  $A[r]$ . Am Ende wird das Pivotelement  $A[r]$  an die richtige Stelle permutiert. Diese Vorgehensweise ist in Algorithmus 39 ausformuliert.

---

**Algorithmus 39** Quicksort

---

**Prozedur** QUICKSORT( $A, l, r$ )

**Falls**  $l < r$  **dann**

$m :=$  TEILEN( $A, l, r$ )

QUICKSORT( $A, l, m - 1$ )

QUICKSORT( $A, m + 1, r$ )

**Ende Falls**

**Ende Prozedur**

**Prozedur** TEILEN( $A, l, r$ )

$x := A[r]$

$i := l$

**Für**  $j = l, \dots, r - 1$

**Falls**  $A[j] \leq x$  **dann**

Vertausche  $A[i]$  mit  $A[j]$

$i := i + 1$

**Ende Falls**

**Ende Für**

Vertausche  $A[i]$  mit  $A[r]$

**Antworte**  $i$

**Ende Prozedur**

---

Die Prozedur TEILEN wählt zunächst als Pivotelement  $x = A[r]$ . Zu Beginn jedes Schleifendurchlaufs gilt:  $i \leq j$ , falls  $k \in \{l, \dots, i - 1\}$  ist  $A[k] \leq x$  und falls  $k \in \{i, \dots, j - 1\}$  ist  $A[k] > x$ . In jedem Schleifendurchlauf wird  $j$  um eins erhöht und gegebenenfalls durch eine Vertauschung von  $A[i]$  mit  $A[j]$  und Inkrementierung von  $i$  diese Invariante beibehalten.

*Beispiel 9.1.* Die Prozedur TEILEN angewandt auf das Datenfeld

3	6	2	7	4	1	8	5
---	---	---	---	---	---	---	---

transformiert dieses in

3	2	4	1	5	7	8	6
---	---	---	---	---	---	---	---

und antwortet mit  $i = 5$ .

Die Laufzeit der Prozedur TEILEN ist  $\Theta(r - l)$ . Die Gesamtlaufzeit von Quicksort hängt also davon ab, wie die Aufteilungen des Datenfeldes stattfinden. Intuitiv ist eine Aufteilung umso besser je schneller die Größe der Instanzen schrumpft. Tatsächlich kann man sich leicht überlegen, dass im Fall einer aufsteigend sortierten Eingabe ein Datenfeld der Länge  $n$  aufgespalten wird in eines der Länge  $n - 1$ , in das Pivotelement, sowie eines der Größe 0. Für diesen

Fall ist die Laufzeit von Quicksort also gegeben durch

$$T(n) = T(n-1) + \Theta(n).$$

Durch die Substitutionsmethode kann leicht gezeigt werden dass  $T(n) = \Theta(n^2)$ . Die Laufzeit von Quicksort im schlechtesten Fall ist also  $\Omega(n^2)$ .

Der obigen Intuition folgend kann man sich vorstellen, dass die beste Aufteilung eines Datenfelds der Länge  $n$  jene ist die zwei Datenfelder der selben Länge erzeugt. Die Laufzeit wird dann (modulo Rundung) beschrieben durch die Rekursionsgleichung

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Nach dem zweiten Fall des Master-Theorems ergibt sich also  $T(n) = \Theta(n \log n)$ .

Die Laufzeit ergibt sich aber selbst dann zu  $O(n \log n)$  wenn eine unbalancierte Aufteilung gewählt wird, so lange die Längen der Teildatenfelder durch eine multiplikative Konstante und die ursprüngliche Länge bestimmt werden. Wird zum Beispiel in jedem Schritt eine 1-zu-9 Aufteilung gewählt, dann erhalten wir (modulo Rundung) die Rekursionsgleichung

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + cn.$$

Im Rekursionsbaum hat dann jede Ebene Kosten  $\leq cn$  und es gibt  $O(\log_{\frac{10}{9}} n)$  Ebenen, insgesamt haben wir also Kosten von  $O(n \log_{\frac{10}{9}} n) = O(n \log n)$ .

Wir analysieren nun den durchschnittlichen Fall. Im durchschnittlichen Fall treten bessere und schlechtere Aufteilungen gemischt auf. Wir gehen, wie immer bisher, davon aus dass alle Permutationen des Eingabedatenfelds gleich wahrscheinlich sind. Weiters nehmen wir zur Vereinfachung der Analyse an, dass die Schlüssel paarweise unterschiedlich sind. Dann können wir zeigen:

**Satz 9.2.** *Die Laufzeit von Quicksort im durchschnittlichen Fall ist  $O(n \log n)$ .*

*Beweis.* Ein Aufruf von Quicksort auf einem Datenfeld der Länge  $n$  führt zu höchstens  $n$  Aufrufen der Prozedur TEILEN. Der Aufwand eines Aufrufs von TEILEN ist  $\Theta(v)$  wobei  $v$  die Anzahl der durchgeführten Vergleiche (" $A[j] \leq x$ ") ist. Weiters wird ja immer mit dem Pivotelement verglichen das in späteren Aufrufen von TEILEN keine Rolle mehr spielt, also wird kein Paar zwei Mal verglichen. Sei  $X$  die Anzahl der Vergleiche die über alle Aufrufe von TEILEN hinweg insgesamt durchgeführt werden. Dann ist die Gesamtlaufzeit von Quicksort  $O(X + n)$ . Wir wollen nun  $EX$  bestimmen.

Seien dazu  $\{z_1, \dots, z_n\}$  die Elemente des Eingabedatenfeldes mit  $z_1 < \dots < z_n$ . Für  $i \leq j$  sei weiters  $Z_{i,j} = \{z_i, \dots, z_j\}$ . Der Algorithmus beginnt mit der Auswahl eines Pivotelements  $z_k$  aus  $Z_{1,n}$  das mit  $z_1, \dots, z_{k-1}, z_{k+1}, \dots, z_n$  verglichen wird um die erste Teilung des Datenfelds zu berechnen. Nach Teilung des Datenfelds werden  $z_1, \dots, z_{k-1}$  sowie  $z_{k+1}, \dots, z_n$  nur noch untereinander verglichen, nicht aber miteinander. Dieser Prozess wiederholt sich dann.

Allgemein können wir also feststellen, dass  $z_i$  mit  $z_j$  verglichen wird genau dann wenn  $z_i$  oder  $z_j$  als erstes in  $Z_{i,j}$  als Pivotelement gewählt wird und damit also kein (früher gewähltes) Pivotelement zwischen  $z_i$  und  $z_j$  steht. Sei

$$X_{i,j} = \begin{cases} 1 & \text{falls } z_i \text{ mit } z_j \text{ verglichen wird} \\ 0 & \text{sonst} \end{cases}$$

Dann ist  $X = \sum_{1 \leq i < j \leq n} X_{i,j}$  und wir erhalten

$$\begin{aligned}
EX &= \sum_{1 \leq i < j \leq n} EX_{i,j} = \sum_{1 \leq i < j \leq n} W(z_i \text{ wird mit } z_j \text{ verglichen}) \\
&= \sum_{1 \leq i < j \leq n} W(z_i \text{ oder } z_j \text{ ist erstes Pivotelement aus } Z_{i,j}) \\
&= \sum_{1 \leq i < j \leq n} \left( \frac{1}{j-i+1} + \frac{1}{j-i+1} \right) \\
&= 2 \sum_{1 \leq i < j \leq n} \frac{1}{j-i+1} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \\
&< 2 \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{1}{k} \\
&= 2 \sum_{i=1}^{n-1} O(\ln n) \\
&= O(n \log n).
\end{aligned}$$

Somit ist auch die Gesamtlaufzeit im durchschnittlichen Fall  $O(n \log n)$ . □

Um diese durchschnittliche Laufzeit als erwartete Laufzeit auch auf ungünstigen Eingaben zu garantieren bietet sich die in Algorithmus 40 angegebene Randomisierung an. Hier wird die

---

**Algorithmus 40** Randomisiertes Quicksort

---

**Prozedur** QUICKSORTRANDOMISIERT( $A, l, r$ )

**Falls**  $l < r$  **dann**

$i := \text{ZUFALL}(l, r)$

Vertausche  $A[i]$  und  $A[r]$

$m := \text{TEILEN}(A, l, r)$

QUICKSORT( $A, l, m-1$ )

QUICKSORT( $A, m+1, r$ )

**Ende Falls**

**Ende Prozedur**

---

Auswahl eines zufälligen Elements als Pivotelement dadurch realisiert, dass das zufällig gewählte Element mit  $A[r]$  vertauscht wird bevor die Teilung des Datenfelds durchgeführt wird. Ein zufällig gewähltes Pivotelement teilt das aktuelle Teildatenfeld mit großer Wahrscheinlichkeit gut auf. Diese randomisierte Variante von Quicksort hat, unabhängig vom Eingabedatenfeld, die erwartete Laufzeit  $O(n \log n)$ .

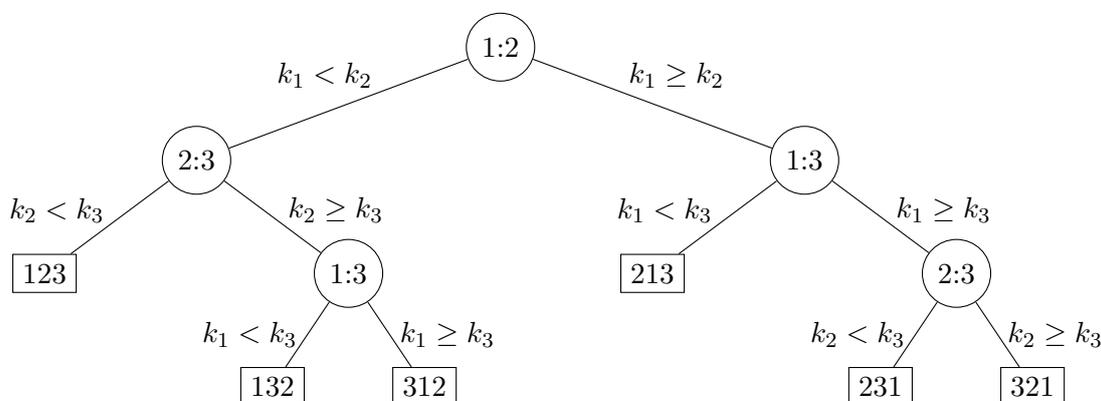
### 9.3 Eine untere Schranke auf Sortieralgorithmen

Wir haben nun einige Sortierverfahren gesehen deren Laufzeit immer mindestens  $n \log n$  war. Es ist natürlich zu fragen, ob es auch besser möglich ist. Eine positive Antwort auf diese Frage bestünde in einem schnelleren Sortierverfahren (das in gewissen Spezialfällen tatsächlich

existiert, siehe unten). Eine negative Antwort darauf zu geben fällt aber viel schwerer da ein Beweis über alle Sortierverfahren notwendig wäre. Wir werden hier eine untere Schranke für ein bestimmte Klasse von Sortierverfahren beweisen: für vergleichsbasierte Sortierverfahren. Diese zeichnen sich dadurch aus, dass sie lediglich Schlüsselvergleiche verwenden können, um die korrekte Reihenfolge der Eingabedaten zu bestimmen (nicht aber z.B. Schlüssel addieren können). Alle bisher in dieser Vorlesung betrachteten Sortierverfahren sind von dieser Art.

Die Entscheidungen die ein vergleichsbasiertes Sortierverfahren trifft können durch einen Entscheidungsbaum dargestellt werden. Bei einer Eingabe von Schlüssel  $k_1, \dots, k_n$  stellt ein Knoten mit Beschriftung  $i:j$  einen Vergleich von  $k_i$  mit  $k_j$  dar. Ein Blatt stellt dann die sortierende Permutation der Eingabe dar. Auf diese Weise induziert ein vergleichsbasiertes Sortierverfahren eine Folge  $(T_n)_{n \geq 2}$  von Entscheidungsbaum. Der Entscheidungsbaum  $T_n$  hat dann  $n!$  Blätter, denn einerseits muss ja jede Permutation vorkommen, da sie als Eingabe möglich ist. Andererseits unterscheiden sich die Permutationen an zwei verschiedenen Blättern und so kann keine Permutation zwei Mal vorkommen.

*Beispiel 9.2.* Ein Entscheidungsbaum eines vergleichsbasierten Sortierverfahrens für  $n = 3$  ist:



Wir können dann zeigen:

**Satz 9.3.** Jedes vergleichsbasierte Sortierverfahren benötigt im schlechtesten Fall Laufzeit  $\Omega(n \log n)$ .

*Beweis.* Wir betrachten ein Eingabedatenfeld der Länge  $n$  und ein Sortierverfahren  $\mathcal{A}$ . Sei  $b$  die Anzahl der Blätter des Entscheidungsbaums von  $\mathcal{A}$  für Eingabe der Länge  $n$ , dann ist  $b = n!$ . Sei  $d$  die maximale Tiefe eines Blattes, dann ist  $b \leq 2^d$  und so erhalten wir  $n! \leq 2^d$ , d.h.  $\log(n!) \leq d$ . Da  $n! \geq (\frac{n}{2})^{\frac{n}{2}} = 2^{\frac{n}{2} \log \frac{n}{2}}$ , ist  $\log(n!) \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$ .  $\square$

Um den Durchschnittsfall zu behandeln beweisen wir zunächst ein Lemma. Für einen Baum  $T$  bezeichnen wir den Durchschnitt der Tiefen der Blätter von  $T$  als *mittlere Tiefe*  $\bar{d}(T)$  von  $T$ .

**Lemma 9.2.** Ein binärer Baum  $T$  mit  $b \geq 2$  Blättern hat mittlere Tiefe  $\bar{d}(T) \geq \log b$ .

*Beweis.* Wir gehen mit Induktion nach  $b$  vor. Der Fall  $b = 2$  ist trivial. Für den Induktionsschritt sei  $T_1$  der linke Teilbaum von  $T$  mit  $b_1$  Blättern und  $T_2$  der rechte Teilbaum von  $T$  mit  $b_2$  Blättern. Dann ist  $b = b_1 + b_2$  und

$$\begin{aligned} \bar{d}(T) &= \frac{1}{b}(b_1(\bar{d}(T_1) + 1) + b_2(\bar{d}(T_2) + 1)) \\ &\stackrel{\text{IH}}{\geq} \frac{1}{b}(b_1(\log b_1 + 1) + b_2(\log b_2 + 1)) \\ &= \frac{1}{b_1 + b_2}(b_1 \log 2b_1 + b_2 \log 2b_2). \end{aligned}$$

Diese Funktion erreicht ihr Minimum bei  $b_1 = b_2 = \frac{b}{2}$ , also

$$\begin{aligned} &\geq \frac{1}{b} \left( \frac{b}{2} \log b + \frac{b}{2} \log b \right) \\ &= \log b. \end{aligned}$$

□

**Satz 9.4.** *Jedes vergleichsbasierte Sortierverfahren benötigt im durchschnittlichen Fall Laufzeit  $\Omega(n \log n)$ .*

*Beweis.* Sei  $\mathcal{A}$  ein vergleichsbasiertes Sortierverfahren und sei  $T_n$  der Entscheidungsbaum von  $\mathcal{A}$  für Eingaben der Länge  $n$ , dann hat  $T_n$  genau  $n!$  Blätter und damit ist, nach Lemma 9.2,  $\bar{d}(T_n) \geq \log n!$ . Wie im Beweis von Satz 9.3 gezeigt ist  $\log n! = \Omega(n \log n)$ . □

Man kann durchaus auch sinnvolle Sortierverfahren angeben, die keine vergleichsbasierten Sortierverfahren sind und, zumindest für gewisse Spezialfälle, eine bessere Laufzeit haben. *Zählsortieren* geht davon aus, dass die Schlüssel in einen bestimmten Bereich  $\{1, \dots, k\}$  fallen und erlaubt dann eine Sortierung in Zeit  $\Theta(n + k)$  wobei  $n$  die Länge des Eingabedatenfelds ist. Für hinreichend kleine  $k$  ist damit also ein Sortierverfahren mit Laufzeit  $\Theta(n)$  angegeben. Ein Einsatz dieses Verfahrens ist nur sinnvoll wenn  $k$  nicht größer als zirka  $n \log n$  ist.

*Beispiel 9.3.* Bei dem Eingabedatenfeld

$$A = \begin{array}{|c|c|c|c|c|c|c|c|} \hline a & b & c & d & e & f & g & h \\ \hline 2 & 4 & 3 & 2 & 1 & 4 & 3 & 3 \\ \hline \end{array}$$

sind alle Schlüssel in  $\{1, \dots, 4\}$ , d.h. also  $k = 4$ . Zunächst zählen wir, für alle  $j \in \{1, \dots, k\}$  wie viele Elemente den Schlüssel  $j$  haben und erhalten.

1	2	3	2
---	---	---	---

Daraus berechnen wir, für  $j \in \{1, \dots, k\}$  den letzten Index mit Schlüssel  $j$  im sortierten Datenfeld.

1	3	6	8
---	---	---	---

Das induziert eine Unterteilung

1	2	2	3	3	3	4	4

des Ausgabedatenfelds. Dieses wird dann, der Unterteilung folgend, mit den Elementen von  $A$  befüllt und wir erhalten

e	a	d	c	g	h	b	f
1	2	2	3	3	3	4	4

Zählsortieren ist in Algorithmus 41 als Pseudocode ausformuliert. Man erkennt leicht, dass diese Prozedur Laufzeit  $\Theta(n + k)$  hat.

---

**Algorithmus 41** Zähler sortieren

---

**Prozedur** ZÄHLSORTIEREN( $A, k$ )Sei  $B[1, \dots, A.Länge]$  ein neues DatenfeldSei  $C[1, \dots, k]$  ein neues Datenfeld, überall mit 0 initialisiert.**Für**  $i := 1, \dots, A.Länge$  $C[A[i].x] := C[A[i].x] + 1$ **Ende Für****Für**  $i := 2, \dots, k$  $C[i] := C[i] + C[i - 1]$ **Ende Für****Für**  $i := A.Länge, \dots, 1$  $B[C[A[i].x]] := A[i]$  $C[A[i].x] := C[A[i].x] - 1$ **Ende Für****Antworte**  $B$ **Ende Prozedur**

---

## 9.4 Primzahltests

Wir wollen uns nun mit dem folgenden Entscheidungsproblem beschäftigen:

<b>Primzahlen</b>
-------------------

Eingabe:  $n \in \mathbb{N}$ Ausgabe: "ja" falls  $n \in \mathbb{P}$ , "nein" sonst

Dieses Problem hat wichtige Anwendungen in der Kryptographie, zum Beispiel bildet es die Grundlage der Schlüsselerzeugung beim RSA-Verfahren das wir in Abschnitt 9.5 genauer besprechen werden. Man kann sich leicht überlegen, dass eine Zahl  $n \in \mathbb{N}$  zusammengesetzt ist genau dann wenn sie einen Teiler  $\leq \sqrt{n}$  hat, so dass es für einen Primzahltest ausreichend ist, Probedivisionen von 1 bis  $\sqrt{n}$  durchzuführen. Dieses auf Probedivisionen basierende Verfahren hat also Laufzeit  $O(\sqrt{n})$ . In der Praxis hat man es aber oft mit Primzahlen zu tun die etwa 1000 bis 3000 Bits haben. Nun ist z.B.  $\sqrt{2^{2000}} = 2^{1000}$  was zu hoch ist, um zu einem praktisch einsetzbaren Algorithmus zu führen.

Seit 2002 ist ein Algorithmus bekannt der in Zeit  $O((\log n)^k)$  für ein  $k \in \mathbb{N}$  entscheidet ob ein gegebenen  $n \in \mathbb{N}$  eine Primzahl ist (AKS-Verfahren). Das derzeit kleinste bekannte solche  $k$  ist  $k = 7$ . Dieses Resultat hat bisher nur theoretische Bedeutung. Für die praktische Anwendung ist diese Klasse von Algorithmen nicht effizient genug. In der Praxis wendet man stattdessen probabilistische Primzahltests an die effizient sind und mit beliebig großer Wahrscheinlichkeit  $< 1$  eine korrekte Antwort liefern. Um diese näher zu besprechen rufen wir uns zunächst den kleinen Satz von Fermat in Erinnerung:

**Satz 9.5** (Kleiner Satz von Fermat). Sei  $n \in \mathbb{P}$ ,  $a \geq 1$ ,  $\text{ggT}(a, n) = 1$ , dann ist

$$a^{n-1} \equiv 1 \pmod{n}.$$

Auf Basis dieses Satz ist ein einfacher Primzahltest möglich: Gegeben  $n \in \mathbb{N}$  wählen wir zufällig ein  $a \in \{2, \dots, n - 1\}$  und überprüfen (mit dem euklidischen Algorithmus) ob  $\text{ggT}(a, n) = 1$ . Ist das nicht der Fall ist  $a$  ein Teiler von  $n$  und damit  $n \notin \mathbb{P}$ . Falls  $\text{ggT}(a, n) = 1$  ist dann

überprüfen wir ob  $a^{n-1} \equiv 1 \pmod{n}$ . Falls  $a^{n-1} \not\equiv 1 \pmod{n}$  dann ist  $n \notin \mathbb{P}$  (auch wenn wir keinen Teiler von  $n$  kennen). In diesem Fall heißt  $a$  auch *Fermat-Zeuge gegen die Primalität von  $n$* . Ist hingegen  $a^{n-1} \equiv 1 \pmod{n}$  dann endet der Algorithmus mit der Vermutung dass  $n \in \mathbb{P}$ . Die Laufzeit dieses Verfahrens beträgt bei Verwendung effizienter Exponentiation  $O(\log n)$ .

Nun hat dieser Primzahltest allerdings einen bedeutenden Defekt: Es gibt zusammengesetzte Zahlen  $n$  so dass für alle  $a \in \{2, \dots, n-1\}$  mit  $\text{ggT}(a, n) = 1$  gilt:  $a^{n-1} \equiv 1 \pmod{n}$ . Solche Zahlen heißen Carmichael-Zahlen. Es gibt unendlich viele Carmichael-Zahlen, die kleinste ist  $3 \cdot 11 \cdot 17 = 561$ . Wenn wir diese aber ausklammern, so können wir zeigen:

**Lemma 9.3.** *Sei  $n \in \mathbb{N}$  eine zusammengesetzte Zahl die keine Carmichael-Zahl ist. Dann gibt es in  $\{1, \dots, n-1\}$  höchstens  $\frac{n-1}{2}$  Zahlen die zu  $n$  teilerfremd sind aber keine Fermat-Zeugen gegen die Primalität von  $n$  sind.*

*Beweis.* Für einen Ring  $R$  sei  $R^\times = \{r \in R \mid \exists s \in R \text{ so dass } rs = 1.\}$ .  $R^\times$  bildet mit der Multiplikation von  $R$  eine Gruppe die als Einheitengruppe von  $R$  bezeichnet wird. Nun ist  $(\mathbb{Z}/n\mathbb{Z})^\times = \{a \in \mathbb{Z}/n\mathbb{Z} \mid \text{ggT}(a, n) = 1\}$ . Wir definieren  $B = \{b \in \mathbb{Z}/n\mathbb{Z} \mid b^{n-1} \equiv 1 \pmod{n}\}$ . Dann ist  $B$  Untergruppe von  $(\mathbb{Z}/n\mathbb{Z})^\times$ . Nach dem Satz von Lagrange gilt also  $|B| \mid |(\mathbb{Z}/n\mathbb{Z})^\times|$ . Da  $n$  zusammengesetzt aber keine Carmichael-Zahl ist gibt es ein  $a \in (\mathbb{Z}/n\mathbb{Z})^\times$  mit  $a^{n-1} \not\equiv 1 \pmod{n}$ . Also ist  $B$  eine echte Untergruppe von  $(\mathbb{Z}/n\mathbb{Z})^\times$  und damit erhalten wir  $|B| \leq \frac{1}{2}|(\mathbb{Z}/n\mathbb{Z})^\times| \leq \frac{n-1}{2}$ .  $\square$

Wenn wir also für ein zusammengesetztes  $n \in \mathbb{N}$  das keine Carmichael-Zahl ist ein  $a \in \{1, \dots, n-1\}$  uniform verteilt wählen, so ist  $W(\text{ggT}(a, n) = 1 \text{ und } a^{n-1} \equiv 1 \pmod{n}) \leq \frac{1}{2}$ . Falls also der Fermattest auf einem  $n \in \mathbb{N}$  das keine Carmichael-Zahl ist mit "vermutlich ist  $n \in \mathbb{P}$ " antwortet, so irrt er sich mit einer Wahrscheinlichkeit von  $\leq \frac{1}{2}$ . Dieser Test kann nun für verschiedene, unabhängig voneinander zufällig gewählte,  $a$  wiederholt werden um so die Fehlerwahrscheinlichkeit beliebig nahe an 1 zu bringen.

Sinnvoll einsetzbar ist der Fermat Test aufgrund der Tatsache dass er auf Carmichael-Zahlen nicht funktioniert allerdings nicht. Ein Test der diesen Defekt nicht hat ist der Miller-Rabin Test. Dieser basiert auf dem folgenden Satz

**Satz 9.6.** *Sei  $n \geq 1$  ungerade, sei  $n-1 = 2^s d$ ,  $d$  ungerade, sei  $a \geq 1$  mit  $\text{ggT}(a, n) = 1$ . Falls  $n \in \mathbb{P}$ , dann ist*

$$a^d \equiv 1 \pmod{n}$$

oder es gibt ein  $r \in \{0, \dots, s-1\}$  so dass

$$a^{2^r d} \equiv -1 \pmod{n}.$$

*Ohne Beweis.*

**Definition 9.1.** Sei  $n \geq 1$  ungerade, sei  $n-1 = 2^s d$ ,  $d$  ungerade. Dann heißt  $a \in \mathbb{N}$  *Miller-Rabin Zeuge gegen die Primalität von  $n$*  falls  $\text{ggT}(a, n) = 1$  und  $a^d \not\equiv 1 \pmod{n}$  sowie  $a^{2^r d} \not\equiv -1 \pmod{n}$  für alle  $r \in \{0, \dots, s-1\}$ :

Gegeben  $n$  und  $a \in \{1, \dots, n-1\}$  kann in Zeit  $O(\log n)$  festgestellt werden ob  $a$  ein Miller-Rabin Zeuge gegen die Primalität von  $n$  ist. Die Bedingung  $\text{ggT}(a, n) = 1$  kann mit dem euklidischen Algorithmus überprüft werden. Die Berechnung von  $a^d$  modulo  $n$  kann durch die auf Binärdarstellung basierende schnelle Exponentiation in Zeit  $O(\log n)$  durchgeführt werden. Die Berechnung der  $a^{2^r d}$  modulo  $n$  geschieht durch sukzessives Quadrieren.

**Satz 9.7.** Sei  $n \geq 3$  eine ungerade zusammengesetzte Zahl, dann gibt es in  $\{1, \dots, n-1\}$  höchstens  $\frac{n-1}{4}$  Zahlen, die zu  $n$  teilerfremd sind aber keine Miller-Rabin Zeugen gegen die Primalität von  $n$  sind.

Ohne Beweis.

---

**Algorithmus 42** Miller-Rabin Primzahltest

---

**Prozedur** MILLER-RABIN( $n, t$ )

**Für**  $i := 1, \dots, t$

$a := \text{ZUFALL}(1, n-1)$

**Falls**  $\text{ggT}(a, n) = 1$  **dann**

**Falls**  $a$  Miller-Rabin Zeuge gegen Primalität von  $n$  ist **dann**

**Antworte** “zusammengesetzt”

**Ende Falls**

**sonst**

**Antworte** “zusammengesetzt”

**Ende Falls**

**Ende Für**

**Antworte** “wahrscheinlich prim”

**Ende Prozedur**

---

**Korollar 9.1.** Sei  $n \geq 3$  ungerade und zusammengesetzt, dann ist die Wahrscheinlichkeit dass MILLER-RABIN( $n, t$ ) mit “wahrscheinlich prim” antwortet höchstens  $2^{-2t}$ .

Ein Primzahltest wie dieser wird zur Erzeugung von großen Primzahlen wie folgt eingesetzt. Man wählt eine zufällige Zahl  $n$  der gewünschten Größenordnung (also z.B. mit 2048 Bits). Für ein geeignetes  $t$  wird dann MILLER-RABIN( $n, t$ ) ausgeführt. Ergibt das die Antwort “wahrscheinlich prim” wird  $n$  als Primzahl betrachtet und zurückgegeben. Ergibt das die Antwort “zusammengesetzt” wiederholt man das Verfahren für eine neue Zufallszahl  $n$ . Ein Wahl von z.B.  $t = 15$  garantiert bereits eine Fehlerwahrscheinlichkeit von höchstens 1 zu einer Milliarde. Das ist für die meisten praktische Anwendungen ausreichend.

## 9.5 Der RSA-Algorithmus

Als Anwendung großer Primzahlen wollen wir das RSA-Verschlüsselungsverfahren betrachten. Das RSA-Verfahren ist ein public-key-Verfahren, d.h. der Schlüssel einer Person X besteht aus zwei Teilen: einem öffentlichen Schlüssel der publiziert wird und zur Verschlüsselung von Nachrichten an X verwendet werden kann und einem privatem Schlüssel der im Besitz von X ist und geheim bleibt. Die Schlüsselerzeugung funktioniert wie folgt: Person X wählt zufällig zwei (vermutete) Primzahlen  $p$  und  $q$  wie im vorherigen Abschnitt beschrieben. Dann wird  $n = pq$  als RSA-Modul bezeichnet. Weiters wählt X eine natürliche Zahl  $e$  mit

$$1 < e < (p-1)(q-1) \text{ und } \text{ggT}(e, (p-1)(q-1)) = 1$$

und<sup>2</sup> berechnet eine Zahl  $d$  mit

$$1 < d < (p-1)(q-1) \text{ und } ed \equiv 1 \pmod{(p-1)(q-1)}$$

---

<sup>2</sup>Das Auftreten von  $(p-1)(q-1)$  erklärt sich durch den Wert der Eulerschen  $\varphi$ -Funktion auf  $n$ :  $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$ .

was mit dem erweiterten euklidischen Algorithmus möglich ist. Dann ist  $(n, e)$  der öffentliche Schlüssel und  $d$  der private Schlüssel.

Will nun eine Person Y eine Nachricht an X schicken so geht sie zur Verschlüsselung wie folgt vor. Zunächst nehmen wir an, dass für die Nachricht  $m$  gilt:  $0 \leq m < n$ . Längere Nachrichten werden in Blöcke unterteilt von denen jeder in  $[0, n[$  zu liegen kommt. Die Nachricht  $m$  wird dann verschlüsselt zu

$$c \equiv m^e \pmod{n}.$$

Um diese Verschlüsselung durchzuführen reicht es, den öffentlichen Schlüssel  $(n, e)$  zu kennen. Die Entschlüsselung basiert dann auf dem folgenden Satz

**Satz 9.8.** *Sei  $(n, e)$  ein öffentlicher Schlüssel, sei  $d$  der dazugehörige private Schlüssel und sei  $0 \leq m < n$ . Dann gilt  $(m^e)^d \equiv m \pmod{n}$ .*

Um diesen Satz zu beweisen machen wir noch kurz die folgende zahlentheoretische Beobachtung.

**Lemma 9.4.** *Seien  $n_1, n_2 \in \mathbb{Z}$  relativ prim, seien  $a, b \in \mathbb{Z}$  mit  $a \equiv b \pmod{n_1}$  und  $a \equiv b \pmod{n_2}$ , dann ist  $a \equiv b \pmod{n_1 n_2}$ .*

*Beweis.* Da  $a \equiv b \pmod{n_1}$  und  $a \equiv b \pmod{n_2}$  gibt es  $k_1, k_2 \in \mathbb{Z}$  so dass  $a - b = n_1 k_1$  und  $a - b = n_2 k_2$ . Also ist  $n_1 k_1 = n_2 k_2$  und da  $n_1$  und  $n_2$  relativ prim sind gilt  $n_2 \mid k_1$ , d.h. also es gibt ein  $l_2 \in \mathbb{Z}$  mit  $k_1 = n_2 l_2$ . Damit ist  $a - b = n_1 n_2 l_2$ , d.h. also  $a \equiv b \pmod{n_1 n_2}$ .  $\square$

*Beweis von Satz 9.8.* Da  $ed \equiv 1 \pmod{(p-1)(q-1)}$  existiert ein  $l \in \mathbb{Z}$  so dass  $ed = 1 + l(p-1)(q-1)$  und damit

$$(m^e)^d = m^{ed} = m^{1+l(p-1)(q-1)} = m(m^{(p-1)(q-1)})^l.$$

Dann gilt

$$(m^e)^d \equiv m(m^{p-1})^{(q-1)l} \equiv m \pmod{p}$$

wegen dem kleinen Satz von Fermat falls  $m$  kein Vielfaches von  $p$  ist. Falls  $m$  ein Vielfaches von  $p$  ist, dann gilt diese Gleichung ebenfalls, da beide Seiten kongruent 0 sind. Analog dazu sieht man dass  $(m^e)^d \equiv m \pmod{q}$ . Aus Lemma 9.4 folgt damit  $(m^e)^d \equiv m \pmod{n}$ .  $\square$

Die Sicherheit dieses Verfahrens basiert darauf, dass  $d$  aus  $n = pq$  und  $e$  nicht, bzw. nur mit unrealistisch großem Aufwand berechnet werden kann. Man beachte aber dass  $d$  sehr wohl effizient aus  $e$  und  $(p-1)(q-1)$  berechnet werden kann. Könnte man also  $n$  schnell genug in  $p$  und  $q$  faktorisieren, wäre das RSA-Verfahren geknackt. Je nach Sicherheitsanforderungen ist es nach aktuellem Stand empfehlenswert für  $n$  eine Zahl mit zirka 500 - 3000 Bits zu wählen. Die beiden Primzahlen  $p$  und  $q$  sollten so gewählt werden, dass sie ungefähr die selbe Größenordnung haben, aber auch nicht zu nahe beieinander sind.

# Kapitel 10

## Lineare Optimierung

### 10.1 Einführung

Viele Optimierungsprobleme bestehen darin eine bestimmte Zielfunktion unter Einhaltung gewisser Bedingungen zu maximieren oder zu minimieren. Falls sowohl die Zielfunktion affin linear ist als auch die Bedingungen affin lineare Gleichungen und Ungleichungen sind, dann spricht man von einem *linearen Optimierungsproblem*. In diesem Kapitel werden wir einige Schritte in das weitläufige Thema der linearen Optimierung machen und beginnen dazu mit einem Beispiel.

*Beispiel 10.1.* Eine Fabrik hat eine Maschine zur Verfügung mit der zwei verschiedene Produkte erzeugt werden können. Der Einsatz der Maschine im kommenden Monat soll geplant werden, wie immer natürlich mit dem Ziel den Gewinn zu maximieren. Dabei unterliegt die Verwendung dieser Maschine gewissen Bedingungen und Einschränkungen, die wir im folgenden beschreiben und formalisieren werden. Sei  $x_1$  die Stückzahl des ersten Produkts und  $x_2$  die Stückzahl des zweiten Produkts. Somit ist also  $x_1 \geq 0$  und  $x_2 \geq 0$ .

1. Mit einer Einheit des ersten Produkts lässt sich 15€ Gewinn erzielen, mit einer Einheit des zweiten Produkts 10€. Wir wollen also

$$15x_1 + 10x_2$$

maximieren.

2. Im kommenden Monat stehen auf dieser Maschine 20000 Arbeitsminuten (also ca. 16h pro Arbeitstag) zur Verfügung. Die Produktion einer Einheit des ersten Produkts benötigt 20 Minuten, einer Einheit des zweiten Produkts 10 Minuten. Also

$$20x_1 + 10x_2 \leq 20000, \text{ d.h. } 2x_1 + x_2 \leq 2000.$$

3. Ein bestehender Vertrag verpflichtet zur Produktion von mindestens 200 Stück des zweiten Produkts, also

$$x_2 \geq 200.$$

4. Die Produktion einer Einheit des ersten Produkts erzeugt 10 Gramm CO<sub>2</sub>, einer Einheit des zweiten Produkts 40 Gramm. Der CO<sub>2</sub>-Ausstoß pro Monat ist auf 38 kg beschränkt. Also

$$10x_1 + 40x_2 \leq 38000, \text{ d.h. } x_1 + 4x_2 \leq 3800$$

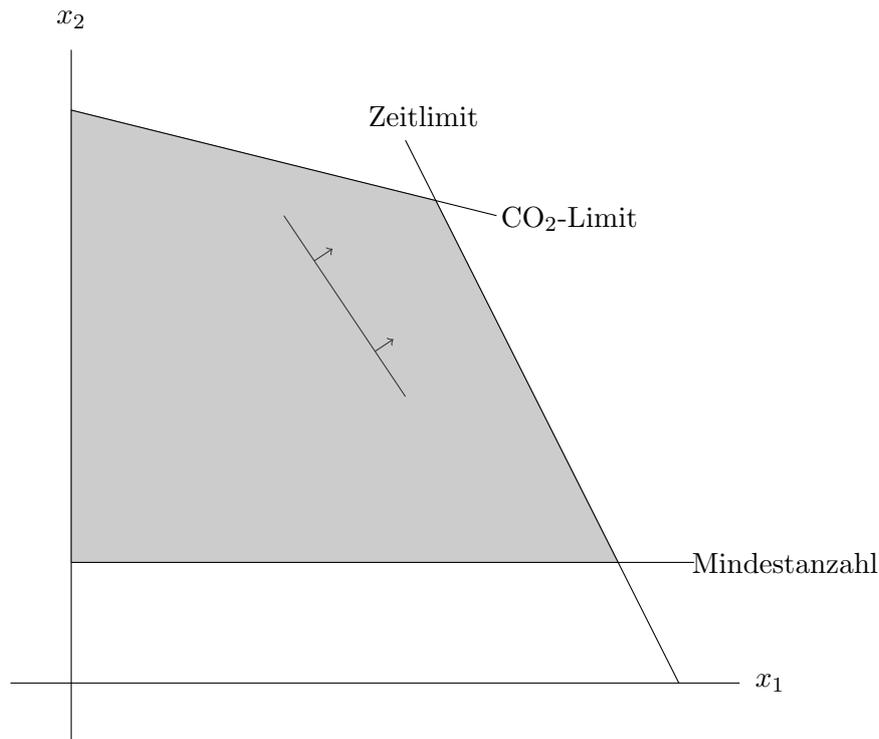


Abbildung 10.1: Beispiel für lineare Optimierung

Wir suchen also  $(x_1, x_2)$  so dass

$$\begin{aligned} 2x_1 + x_2 &\leq 2000 \\ x_2 &\geq 200 \\ x_1 + 4x_2 &\leq 3800 \end{aligned}$$

und, unter allen Paaren die diese Bedingungen erfüllen, soll  $15x_1 + 10x_2$  maximal sein. Die Situation ist in [Abbildung 10.1](#) graphisch dargestellt. Der graue Bereich besteht aus jenen  $(x_1, x_2)$  die alle Bedingungen erfüllen. Ein maximaler Punkt kann darin auf geometrische Weise wie folgt gefunden werden: Wir betrachten die Gerade  $z = 15x_1 + 10x_2$  für wachsendes  $z$  und schieben sie auf diese Weise durch den zulässigen Bereich bis jede weitere Verschiebung um ein  $\varepsilon > 0$  zu einem leeren Schnitt mit dem zulässigen Bereich führen würde. Hier geschieht das, wenn die Gerade den Schnittpunkt der Zeit- mit der CO<sub>2</sub>-Gerade enthält. Dieser ist, wie man leicht anhand der Geradengleichungen berechnen kann,  $(x_1, x_2) = (600, 800)$ . Der maximale Gewinn wird also erreicht bei Erzeugung von 600 Einheiten des ersten Produkts und 800 Einheiten des zweiten Produkts und beträgt somit 17.000€.

Ausgehend von diesem Beispiel betrachten wir nun die allgemeine Problemstellung. Eine *affine Gleichung* ist ein Ausdruck der Form  $\sum_{i=1}^n a_i x_i = b$  wobei  $a_1, \dots, a_n, b \in \mathbb{R}$  und  $x_1, \dots, x_n$  reellwertige Variablen sind. Eine *affine Ungleichung* ist ein Ausdruck der Form  $\sum_{i=1}^n a_i x_i \leq b$  wobei  $a_1, \dots, a_n, b \in \mathbb{R}$  und  $x_1, \dots, x_n$  reellwertige Variablen sind. Eine *affine Funktion*  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  ist gegeben durch  $f(x) = \sum_{i=1}^n a_i x_i + b$  wobei  $a_1, \dots, a_n, b \in \mathbb{R}$ .

**Definition 10.1.** Ein *lineares Programm* in den Variablen  $x_1, \dots, x_n$  besteht aus einer endlichen Menge  $E$  von affinen Gleichungen und affinen Ungleichungen sowie einer affinen Funktion  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  und der Information ob  $f$  maximiert oder minimiert werden soll. Ein Vektor  $x \in \mathbb{R}^n$

heißt *zulässige Lösung* falls  $x$  die Bedingungen  $E$  erfüllt. Eine zulässige Lösung  $x$  heißt *optimale Lösung* falls  $f(x)$  optimal unter allen zulässigen Lösungen ist.

Die Funktion  $f$  wird auch als *Zielfunktion* bezeichnet. Für ein  $x \in \mathbb{R}^n$  wird  $f(x)$  auch als *Zielwert von  $x$*  bezeichnet. Zunächst kann man sich überlegen dass für ein gegebenes lineares Programm  $E, f$  einer der folgenden Fälle zutrifft.

1.  $E, f$  besitzt eine optimale Lösung mit einem Zielwert  $z \in \mathbb{R}$ . Das ist der interessanteste Fall der auch in Beispiel 10.1 auftritt. Man beachte, dass in diesem Fall zwar der Zielwert eindeutig ist, die optimale Lösung aber im Allgemeinen nicht.
2.  $E, f$  besitzt keine zulässigen Lösungen. In diesem Fall heißt  $E, f$  *unlösbar*. Ein Beispiel für diesen Fall erhielte man aus Beispiel 10.1 wenn (etwa aus einem weiteren Vertrag) eine Mindeststückzahl des ersten Produkts von z.B. 1000 Stück gefordert wäre.
3.  $E, f$  besitzt Lösungen mit beliebig großem (für Maximierung) bzw. beliebig kleinem (bei Minimierung) Zielwert. In diesem Fall heißt  $E, f$  *unbeschränkt*. Dieser Fall würde auftreten, wenn man in Beispiel 10.1 die limitierenden Zeit- und CO<sub>2</sub>-Beschränkungen weglassen würde.

Wir können also das folgende Berechnungsproblem formulieren.

<p><b>Lineare Optimierung</b></p> <p>Eingabe: ein lineares Programm <math>E, f</math></p> <p>Ausgabe: “unbeschränkt” falls <math>E, f</math> unbeschränkt ist, “unlösbar” falls <math>E, f</math> unlösbar ist und eine optimale Lösung <math>x \in \mathbb{R}^n</math> von <math>E, f</math> sonst</p>
---

Es gibt polynomiale Algorithmen die dieses Problem lösen. Wir werden in Abschnitt 10.3 den Simplex-Algorithmus besprechen. Dieser ist zwar nicht polynomial, praktisch allerdings recht effizient, da für den Simplex-Algorithmus schlechte Eingaben in der Praxis selten sind. Außerdem ist der Simplex-Algorithmus in gewissem Sinn eine Verallgemeinerung des gaußschen Eliminationsverfahrens und alleine deswegen schon interessant.

## 10.2 Reduktionen auf lineare Optimierung

Auf den ersten Blick erweckt das Problem der linearen Optimierung vielleicht den Eindruck recht spezifisch für diese Art von Produktionsplanung und verwandte Situationen zu sein. In der Tat ist das Problem aber sehr allgemein in dem Sinn dass für viele Probleme recht direkte Reduktionen auf lineare Optimierung existieren. Um das zu illustrieren werden wir hier einige solche Reduktionen angeben.

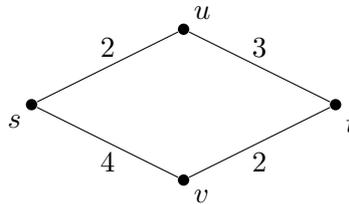
**Kürzester Pfad.** Gegeben ein zusammenhängender Graph  $G = (V, E)$  und eine Kostenfunktion  $c : E \rightarrow \mathbb{R}_{\geq 0}$  sowie einen Startknoten  $s \in V$  und einen Zielknoten  $t \in V$  wollen wir die Länge des kürzesten (d.h. billigsten) Pfades von  $s$  nach  $t$  bestimmen. In Abschnitt 7.3 haben wir den Algorithmus von Dijkstra kennengelernt, der einen solchen Pfad berechnet. Um dieses Problem als lineares Programm darzustellen führen wir für jedes  $v \in V$  eine reellwertige Variable  $d_v$  ein, die die Länge des kürzesten Pfades von  $s$  nach  $v$  darstellen soll. Dann gilt

$$d_s = 0 \quad \text{und}$$

$$d_v \leq d_u + c(\{u, v\}) \quad \text{für alle } \{u, v\} \in E.$$

Wenn wir unter diesen Bedingungen  $d_t$  maximieren, stellen wir sicher, dass implizit ein Pfad von  $s$  nach  $t$  ausgewählt wird, da für jedes  $d_v$  für  $v = t, \dots, s$  eine der Ungleichungen mit Gleichheit erfüllt wird. Jeder Knoten  $u$  der das erreicht ist der  $v$  vorausgehende auf einem kürzesten Pfad.

*Beispiel 10.2.* Der Graph



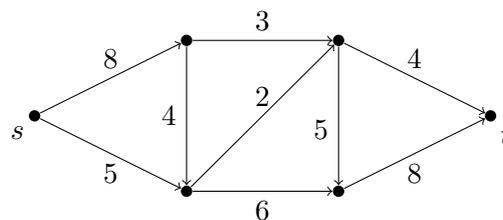
führt zu dem linearen Programm

$$\begin{aligned} d_t \text{ max!} \\ d_s = 0, d_u \leq d_s + 2, d_v \leq d_s + 4, d_t \leq d_u + 3, d_t \leq d_v + 2. \end{aligned}$$

von dem  $(d_s, d_u, d_v, d_t) = (0, 2, 0, 5)$  eine optimale Lösung ist.

**Maximaler Fluss.** Wir betrachten jetzt das Problem des maximalen Flusses (in einem Transportnetzwerk). Das modellieren wir wie folgt: Wir gehen von einem gerichteten, zusammenhängenden Graphen  $G = (V, E)$  aus sowie von einem ausgezeichneten Quellknoten  $s \in V$  und einem ausgezeichneten Zielknoten  $t \in V$ . Weiters hat jede Kante eine gewisse (Transport-)kapazität die durch eine Funktion  $c : E \rightarrow \mathbb{R}_{\geq 0}$  angegeben wird. Es gibt an den Knoten außer den im Graph angegebenen Kanten keine Möglichkeit der Entnahme oder Einspeisung. Weiters ist nirgendwo eine Lagerung möglich. Wir wollen so viel wie möglich von  $s$  nach  $t$  transportieren. Als Anwendungsbeispiel können wir uns etwa den Transport von Rohöl durch ein Netzwerk von Pipelines vorstellen.

*Beispiel 10.3.* Ein Beispiel für einen gerichteten Graphen mit Kapazitätsbeschriftungen auf den Kanten ist:



**Definition 10.2.** Sei  $G = (V, E)$  ein gerichteter Graph,  $s, t \in V$  und  $c : E \rightarrow \mathbb{R}_{\geq 0}$  eine Kapazitätsfunktion. Ein Fluss für  $G$  und  $e$  ist dann eine Funktion  $f : E \rightarrow \mathbb{R}_{\geq 0}$  so dass

1.  $f(e) \leq c(e)$  für alle  $e \in E$  und
2.  $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$  für alle  $v \in V \setminus \{s, t\}$ .

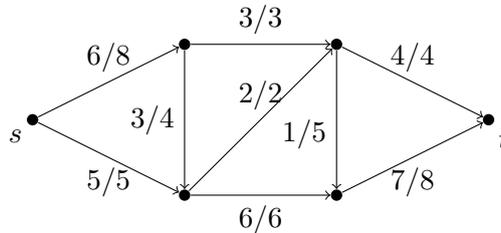
Das Problem der Bestimmung eines maximalen Flusses kann dann wie folgt formuliert werden.

### Maximaler Fluss

Eingabe: gerichteter zusammenhängender Graph  $(V, E)$ ,  $s, t \in V$ , Kapazitätsfunktion  $c : E \rightarrow \mathbb{R}_{\geq 0}$

Ausgabe: ein Fluss  $f : E \rightarrow \mathbb{R}_{\geq 0}$  so dass  $\sum_{(v,t) \in E} f(v, t)$  maximal ist

*Beispiel 10.4.* Ein maximaler Fluss des Graphen aus Beispiel 10.3 ist:



Zur Lösung dieses Problems mittels linearer Optimierung stellen wir einen Fluss  $f$  dar indem wir für jedes  $e \in E$  eine reellwertige Variable  $f_e$  einführen, die  $f(e)$  darstellen soll. Das lineare Programm ist dann:

$$\begin{aligned}
 &0 \leq f_e \quad \text{für alle } e \in E \\
 &f_e \leq c(e) \quad \text{für alle } e \in E \\
 &\sum_{(u,v) \in E} f_{u,v} = \sum_{(v,w) \in E} f_{v,w} \quad \text{für alle } v \in V \setminus \{s, t\} \\
 &\sum_{(v,t) \in E} f_{v,t} \text{ max!}
 \end{aligned}$$

Es ist leicht zu sehen dass eine optimale Lösung dieses linearen Programms ein maximaler Fluss ist.

**Schaltkreisevaluierung.** Ein letztes Problem das wir nun auf lineare Optimierung reduzieren wollen ist die Schaltkreisevaluierung. Das Problem der Schaltkreisevaluierung ist **P**-vollständig, d.h., informell gesagt, dass jedes (mit einem beliebigen Algorithmus) in polynomialer Zeit lösbares Problem durch ein Verfahren für Schaltkreisevaluierung in polynomialer Zeit gelöst werden kann<sup>1</sup>. Diese Reduktion zeigt also, dass auch lineare Optimierung **P**-vollständig ist,

**Definition 10.3.** Ein Boolescher Schaltkreis ist ein endlicher gerichteter azyklischer Graph  $G = (V, E)$  mit einer Knotenbeschriftung  $l : V \rightarrow \{\mathbf{w}, \mathbf{f}, \wedge, \vee, \neg\}$  so dass  $d^-(v) \leq 2$  für alle  $v \in V$  und:

1. Falls  $l(v) \in \{\mathbf{w}, \mathbf{f}\}$ , dann ist  $d^-(v) = 0$ .
2. Falls  $l(v) = \neg$ , dann ist  $d^-(v) = 1$ .
3. Falls  $l(v) \in \{\wedge, \vee\}$ , dann ist  $d^-(v) = 2$ .

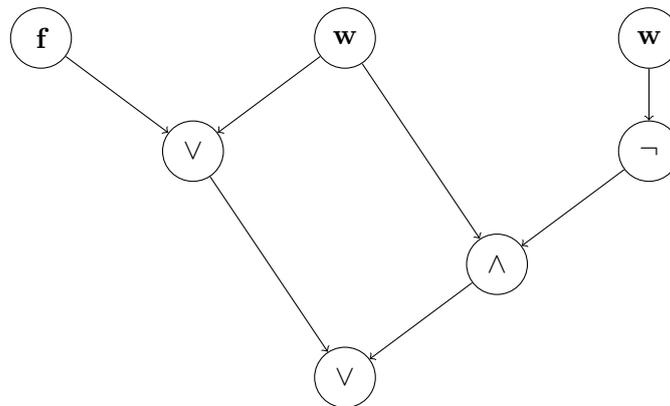
<sup>1</sup>Wir geben hier keine formale Definition von **P**-Vollständigkeit und verweisen stattdessen auf Quellen über Komplexitätstheorie.

Hier steht **w** für “wahr” und **f** für “falsch”. Die logischen Operatoren  $\wedge$ ,  $\vee$  und  $\neg$  werden ihre übliche Bedeutung erhalten. Ein Knoten  $v$  eines Schaltkreises mit  $d^+(v) = 0$  heißt *Ausgabeknoten*. Für jeden Knoten  $v$  wird jetzt induktiv sein Wert  $\llbracket v \rrbracket$  der Semantik der Booleschen Logik folgend definiert: Falls  $l(v) = \mathbf{w}$ , dann  $\llbracket v \rrbracket = 1$ . Falls  $l(v) = \mathbf{f}$ , dann  $\llbracket v \rrbracket = 0$ . Falls  $l(v) \in \{\neg, \wedge, \vee\}$ , dann ergibt sich  $\llbracket v \rrbracket$  aus den folgenden Tabellen:

$a$	$\neg a$
0	1
1	0

$a$	$b$	$a \wedge b$	$a \vee b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

*Beispiel 10.5.* Der Schaltkreis

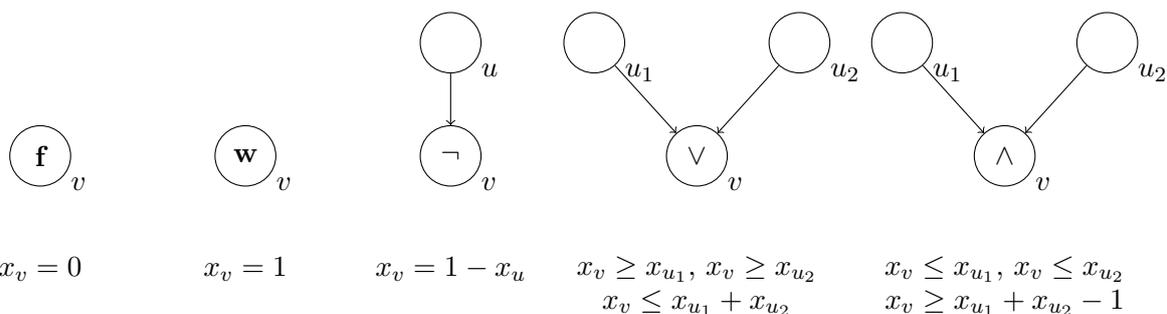


hat einen einzigen Ausgabeknoten  $v$  dessen Wert  $\llbracket v \rrbracket = 1$  ist.

Das Berechnungsproblem der Schaltkreisevaluierung ist dann wie folgt:

**Schaltkreisevaluierung**  
 Eingabe: Ein Schaltkreis mit genau einem Ausgabeknoten  
 Ausgabe: Der Wert des Ausgabeknotens

Dieses Problem wird durch das folgende lineare Programm gelöst. Wir führen für jedes  $v \in V$  eine Variable  $x_v$  ein, die den Wert von  $v$  repräsentieren soll. Für jedes  $x_v$  werden die Ungleichungen  $0 \leq x_v$  und  $x_v \leq 1$  aufgenommen. Für jeden Knoten werden weitere Ungleichungen wie folgt aufgenommen:



Dann lässt sich mit Induktion über den Schaltkreis nachweisen dass die einzige zulässige Lösung dieses linearen Programms jene ist mit  $x_v = \llbracket v \rrbracket$  für alle  $v \in V$ . Insbesondere gilt dadurch  $x_v \in \{0, 1\}$ . Deshalb ist in diesem Fall auch irrelevant ob und was minimiert oder maximiert wird.

### 10.3 Der Simplex-Algorithmus

Wir behandeln jetzt einen Algorithmus zur Lösung des Problems der linearen Optimierung. Dazu ist es sinnvoll, den Formalismus ein wenig einzuschränken.

**Definition 10.4.** Ein *lineares Programm in Standardform* besteht aus einer Matrix  $A \in \mathbb{R}^{m \times n}$ , einem Vektor  $b \in \mathbb{R}^m$  und einem Vektor  $c \in \mathbb{R}^n$ .

Die Matrix  $A$  gemeinsam mit dem Vektor  $b$  spezifiziert  $E$  durch die Ungleichungen  $Ax \leq b$ . Der Vektor  $c$  spezifiziert  $f$  als  $x \mapsto c^T x$ . Zusätzlich schränken wir uns bei einem linearen Programm in Standardform darauf ein  $f$  zu maximieren und nur solche Lösungen zu betrachten wo  $x_i \geq 0$  für alle  $i \in \{0, \dots, n\}$ . Wir definieren also

**Definition 10.5.** Sei  $A, b, c$  ein lineares Programm in Standardform. Ein  $x \in \mathbb{R}^n$  heißt *zulässige Lösung* falls  $Ax \leq b$  und  $x_i \geq 0$  für alle  $i \in \{0, \dots, n\}$ . Eine zulässige Lösung  $x \in \mathbb{R}^n$  heißt *optimale Lösung* falls  $c^T x$  maximal unter allen zulässigen Lösungen ist.

<p><b>Lineare Optimierung (Standardform)</b></p> <p>Eingabe: ein lineares Programm <math>A, b, c</math> in Standardform</p> <p>Ausgabe: “unbeschränkt” falls <math>A, b, c</math> unbeschränkt ist, “unlösbar” falls <math>A, b, c</math> unlösbar ist, eine optimale Lösung <math>x \in \mathbb{R}^n</math> von <math>A, b, c</math> sonst</p>
---

**Definition 10.6.** Zwei max-lineare Programme  $E, f$  und  $E', f'$  heißen *äquivalent* falls:

1. Für jede zulässige Lösung  $x$  von  $E$  existiert eine zulässige Lösung  $x'$  von  $E'$  mit  $f(x) = f'(x')$ .
2. Für jede zulässige Lösung  $x'$  von  $E'$  existiert eine zulässige Lösung  $x$  von  $E$  mit  $f(x) = f'(x')$ .

Ein min-lineares Programm  $E, f$  und ein max-lineares Programm  $E', f'$  heißen *äquivalent* falls:

1. Für jede zulässige Lösung  $x$  von  $E$  existiert eine zulässige Lösung  $x'$  von  $E'$  mit  $f(x) = -f'(x')$ .
2. Für jede zulässige Lösung  $x'$  von  $E'$  existiert eine zulässige Lösung  $x$  von  $E$  mit  $f(x) = -f'(x')$ .

**Lemma 10.1.** *Jedes lineare Programm kann in ein äquivalentes lineares Programm in Standardform transformiert werden.*

*Beweis.* Ein lineares Programm kann sich von einem linearen Programm in Standardform in den folgenden Aspekten unterscheiden: 1. Falls die Zielfunktion  $f$  einen konstanten Teil  $d \in \mathbb{R}$  hat, dann fügen wir eine neue Variable  $v$  sowie die Gleichung  $v = d$  hinzu und ersetzen  $d$

in  $f$  durch  $v$ . Das dadurch entstehende lineare Programm ist äquivalent zum ursprünglichen. 2. Falls die Zielfunktion  $f$  minimiert werden soll, wird stattdessen  $-f$  maximiert. Diese beiden linearen Programme sind dann äquivalent. 3. Eine Variable  $v$  ohne die Bedingung  $v \geq 0$  wird entfernt und durch zwei neue Variablen  $v^+$  und  $v^-$  simuliert, indem jedes Vorkommen von  $v$  durch  $v^+ - v^-$  ersetzt wird. Weiters werden die Bedingungen  $v^+ \geq 0$  und  $v^- \geq 0$  aufgenommen. Diese beiden linearen Programme sind äquivalent da jedes  $a \in \mathbb{R}$  geschrieben werden kann als  $a = a^+ - a^-$  mit  $a^+, a^- \geq 0$ . 4. Eine Gleichung der Form  $f(\bar{x}) = g(\bar{x})$  für zwei affine Funktionen  $f$  und  $g$  wird ersetzt durch die beiden Ungleichungen  $f(\bar{x}) \leq g(\bar{x})$  und  $g(\bar{x}) \leq f(\bar{x})$ . Diese beiden linearen Programme sind äquivalent. Schließlich werden die affinen Ungleichungen in der Form  $Ax \leq b$  geschrieben und die Zielfunktion  $f$  als  $x \mapsto c^T x$ .  $\square$

Wir machen nun einige grundlegende Beobachtungen über die Situation in deren Rahmen der Simplex-Algorithmus motiviert und geometrisch erklärt werden kann.

Zunächst beobachten wir dass jede der Ungleichungen eines linearen Programms in Standardform einen Halbraum von  $\mathbb{R}^n$  definiert. Die Menge der zulässigen Lösungen ist also ein Schnitt von Halbräumen und damit ein *konvexes Polytop*.

Die zu maximierende Funktion  $x \mapsto c^T x$  induziert für festes  $z \in \mathbb{R}$  die Hyperebene  $z = c^T x$ . Der Maximierung von  $z$  entspricht eine Verschiebung der Hyperebene durch das Polytop der zulässigen Lösungen (vgl. dazu auch Beispiel 10.1 im  $\mathbb{R}^2$ ). Falls das lineare Programm lösbar ist, endet diese Verschiebung mit einem  $z$  das eine Hyperebene definiert, die ein  $k$ -dimensionales Unterpolytop enthält, also einen Knoten, eine Kante, eine Seitenfläche, etc. des Polytops der zulässigen Lösungen<sup>2</sup>. Ein Unterpolytop enthält aber mindestens einen Knoten. Wir erkennen also: *Falls das lineare Programm lösbar ist, dann ist ein Knoten des Polytops eine optimale Lösung.*

Es reicht also, die Hyperebene von Knoten zu Knoten zu schieben. Die *Grundidee* für den Simplex-Algorithmus ist nun: wir starten an einem beliebigen Knoten des Polytops. In jedem Schritt bewegen wir uns zu einem benachbarten Knoten an dem der Zielwert mindestens so groß wie beim vorherigen ist.

Zur algebraischen Realisierung dieses Verfahrens führen wir zunächst sogenannte Schlupfvariablen ein. Damit wird eine Ungleichung der Form

$$\sum_{i=1}^k a_i x_i \leq b$$

transformiert zu

$$s = b - \sum_{i=1}^k a_i x_i \quad \text{und} \quad s \geq 0.$$

Dabei wird  $s$  als *Schlupfvariable* bezeichnet. Variablen die keine Schlupfvariablen sind, werden als *freie Variablen* bezeichnet.

**Definition 10.7.** Ein lineares Programm in *Schlupfform* in den Variablen  $\{x_1, \dots, x_n\}$  ist gegeben durch die Menge  $S$  der Indizes der Schlupfvariablen, die Menge  $F$  der Indizes der

---

<sup>2</sup>Falls das lineare Programm unlösbar ist, hat diese Verschiebung keinen Anfang, falls es unbeschränkt ist keine Ende.

freien Variablen mit  $S \cap F = \emptyset$  und  $S \cup F = \{1, \dots, n\}$  sowie den Gleichungen

$$z = \nu + \sum_{j \in F} c_j x_j \text{ max!}$$

$$x_i = b_i - \sum_{j \in F} a_{i,j} x_j \text{ für } i \in S$$

und inkludiert implizit die Nichtnegativitätsbedingungen  $x_1 \geq 0, \dots, x_n \geq 0$ .

Ein lineares Programm in Schlupfform ist also durch das Tupel  $(F, S, A, b, c, \nu)$  vollständig spezifiziert.

**Lemma 10.2.** *Jedes lineare Programm in Standardform kann in ein äquivalentes lineares Programm in Schlupfform transformiert werden.*

*Beweis.* Durch wiederholte Einführung von Schlupfvariablen werden zum ursprünglichen Programm äquivalente lineare Programm erzeugt. Dieser Vorgang endet wenn nur noch Ungleichungen von der Form  $x_i \geq 0$  übrig sind.  $\square$

**Definition 10.8.** Sei  $L = F, S, A, b, c, \nu$  ein lineares Programm in Schlupfform in den Variablen  $\{x_1, \dots, x_n\}$ . Dann gibt es genau ein  $x \in \mathbb{R}^n$  so dass  $x_i = 0$  für alle  $i \in F$  und  $x$  alle Gleichungen erfüllt. Dieses  $x \in \mathbb{R}^n$  wird als *Basislösung* von  $L$  bezeichnet. Falls  $x$  auch alle Nichtnegativitätsbedingungen erfüllt, dann heißt  $x$  *zulässige Basislösung*.

Über seine Basislösung spezifiziert eine Schlupfform also einen bestimmten Punkt im  $\mathbb{R}^n$ . Im Simplex-Verfahren werden wir (nach einer Phase der Initialisierung) nur solche Schlupfformen betrachten deren Basislösung zulässig ist. Auf diese Weise spezifiziert also eine Schlupfform einen Knoten des Polytops der zulässigen Lösungen. Der Simplex-Algorithmus realisiert nun die Bewegung von einem Knoten zu einem Nachbarknoten durch Transformation einer Schlupfform mit zulässiger Basislösung in eine andere Schlupfform mit zulässiger Basislösung. Diese Transformation geschieht durch Austausch einer freien Variablen  $x_f$  mit einer Schlupfvariablen  $x_s$ . In der neuen Basislösung ist dann nicht mehr  $x_f = 0$ , sondern  $x_s = 0$ . Wir illustrieren die Funktionsweise des Simplex-Algorithmus auf der algebraischen Ebene zunächst anhand eines Beispiels.

*Beispiel 10.6.* Wir wollen auf das folgende lineare Programm in Standardform den Simplex-Algorithmus anwenden:

$$z = x_1 + 2x_2 \text{ max!}$$

$$3x_1 + x_2 \leq 15$$

$$x_1 + 4x_2 \leq 16$$

Dieses lineare Programm ist in Abbildung 10.2 graphisch dargestellt. Zunächst bringen wir es in Schlupfform:

$$z = x_1 + 2x_2$$

$$x_3 = 15 - 3x_1 - x_2$$

$$x_4 = 16 - x_1 - 4x_2$$

wobei  $F = \{1, 2\}$ ,  $S = \{3, 4\}$  und  $x_1, x_2, x_3, x_4 \geq 0$  implizit ist. Die Basislösung dieses Programms ist  $(0, 0, 15, 16)$ . Der Vektor  $(0, 0, 15, 16)$  ist eine zulässige Basislösung. Seine Einschränkung auf die ersten beiden Komponenten ist der Vektor  $(x_1, x_2) = (0, 0)$  der den aktuellen Knoten des Polytops im ursprünglichen Problem angibt, siehe Abbildung 10.2.

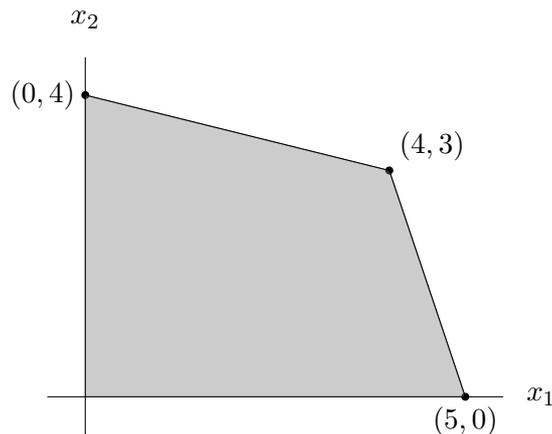


Abbildung 10.2: Beispiel für den Simplex-Algorithmus

Da  $x_2$  in der Zielfunktion den höchsten positiven Koeffizienten hat, wählen wir  $x_2$  zum Vertauschen mit einer Schlupfvariablen. Aus der Gleichung von  $x_3$  folgt dass  $x_2 \leq 15$  ist. Aus der Gleichung von  $x_4$  folgt dass  $x_2 \leq 4$  sein muss. Da beide dieser Bedingungen erfüllt sein müssen muss  $x_2 \leq 4$  sein. In Hinblick darauf, in der nächsten Basislösung  $x_2 = 4$  und damit  $x_4 = 0$  zu setzen vertauschen wir also  $x_2$  und  $x_4$ . Dazu schreiben wir zunächst die Gleichung von  $x_4$  zu  $x_2 = 4 - \frac{1}{4}x_1 - \frac{1}{4}x_4$  um und ersetzen dadurch in allen anderen Gleichungen  $x_2$ . So erhalten wir das neue lineare Programm

$$\begin{aligned} z &= 8 + \frac{1}{2}x_1 - \frac{1}{4}x_4 \\ x_2 &= 4 - \frac{1}{4}x_1 - \frac{1}{4}x_4 \\ x_3 &= 11 - \frac{11}{4}x_1 + \frac{1}{4}x_4 \end{aligned}$$

wobei  $F = \{1, 4\}$  und  $S = \{2, 3\}$  ist. Die Basislösung dieses Programms ist  $(0, 4, 11, 0)$  was dem Punkt  $(x_1, x_2) = (0, 4)$  entspricht, einem benachbarten Knoten des Polytops der zulässigen Lösungen.

Nun hat  $x_1$  in der Zielfunktion den höchsten positiven Koeffizienten. Aus der Gleichung von  $x_2$  folgt dass  $x_1 \leq 16$  ist. Aus der Gleichung von  $x_3$  folgt dass  $x_1 \leq 4$  ist. Wir vertauschen also  $x_1$  und  $x_3$ . Wir haben  $x_1 = 4 - \frac{4}{11}x_3 + \frac{1}{11}x_4$  und damit erhalten wir das neue Programm

$$\begin{aligned} z &= 10 - \frac{4}{22}x_3 - \frac{9}{44}x_4 \\ x_1 &= 4 - \frac{4}{11}x_3 + \frac{1}{11}x_4 \\ x_2 &= 3 + \frac{1}{11}x_3 - \frac{12}{44}x_4 \end{aligned}$$

wobei  $F = \{3, 4\}$ ,  $S = \{1, 2\}$ . Die Basislösung dieses Programms ist  $(4, 3, 0, 0)$  was dem Punkt  $(x_1, x_2) = (4, 3)$  entspricht.

Da nun alle Koeffizienten in der Zielfunktion negativ sind, kann durch Erhöhung der freien Variablen der Zielwert nicht mehr erhöht werden. Wir haben also eine optimale Lösung gefunden:  $x_1 = 4$ ,  $x_2 = 3$  und  $z = 10$ .

---

**Algorithmus 43** Der Simplex-Algorithmus

---

**Prozedur**  $\text{SIMPLEX}(A, b, c)$   
 $(F, S, A, b, c, \nu) := \text{INITSIMPLEX}(A, b, c)$   
**Solange**  $\exists j \in F$  mit  $c_j > 0$   
    Wähle  $f \in F$  mit  $c_f > 0$   
     $\Delta := \text{BESCHRÄNKUNGEN}(S, A, b, f)$   
    Sei  $s \in S$  so dass  $\Delta[s]$  minimal ist  
    **Falls**  $\Delta[s] = \infty$  **dann**  
        **Antworte** “unbeschränkt”  
    **sonst**  
         $(F, S, A, b, c, \nu) := \text{AUSTAUSCH}(F, S, A, b, c, \nu, s, f)$   
    **Ende Falls**  
**Ende Solange**  
**Antworte**  $\text{BASISLÖSUNG}(F, S, A, b, c, \nu)$   
**Ende Prozedur**

---

Der Simplex-Algorithmus kann als Pseudocode wie in Algorithmus 43 formuliert werden. Dabei werden die folgenden Unterprozeduren verwendet:

Die Prozedur  $\text{INITSIMPLEX}$  transformiert ein gegebenes lineares Programm in Standardform in ein Äquivalentes in Schlupfform dessen Basislösung zulässig ist oder liefert “unlösbar” zurück falls das Eingabeprobem unlösbar ist.

Die Prozedur  $\text{BESCHRÄNKUNGEN}(S, A, b, f)$  liefert ein Datenfeld  $\Delta$  der Länge  $n = |F| + |S|$  zurück so dass, für alle  $s \in S$  der Wert  $\Delta[s]$  angibt welchen Wert  $x_f$  maximal annehmen kann, ohne die Schlupfvariable  $x_s$  kleiner als 0 zu machen.

Die Prozedur  $\text{AUSTAUSCH}(F, S, A, b, c, \nu, s, f)$  löst die Gleichung von  $x_s$  nach  $x_f$  auf und setzt diese Darstellung von  $x_f$  in alle anderen Gleichungen ein. Dadurch erzeugt sie eine neue Schlupfform die sich von  $F, S, A, b, c, \nu$  dadurch unterscheidet dass  $s$  von einer Schlupfvariable zu einer freien Variable geworden ist und  $f$  von einer freien Variable zu einer Schlupfvariable. Diese neue Schlupfform hat die selbe Menge zulässiger Lösungen und die selbe Zielfunktion, lediglich deren Darstellung wurde verändert. Diese neue Schlupfform wird dann von  $\text{AUSTAUSCH}$  zurückgegeben.

Zunächst wollen wir die Initialisierung außer acht lassen und damit beginnen die partielle Korrektheit (d.h. ohne Betrachtung der Termination) der Simplex-Schleife zu zeigen. Mit Simplex-Schleife ist hier die Schleife in Algorithmus 43 gemeinsam mit der letzten Zeile, die mit der aktuellen Basislösung antwortet, gemeint. Dazu stellen wir zuerst noch eine Überlegung zur optimalen Lösung an.

**Definition 10.9.** Sei  $A \subseteq \mathbb{R}^n$  und  $f : A \rightarrow \mathbb{R}$ . Ein  $x \in A$  heißt (*globales*) *Optimum* von  $f$  falls  $f(y) \leq f(x)$  für alle  $y \in A$ . Ein  $x \in A$  heißt *lokales Optimum* von  $f$  falls es eine Umgebung  $U$  von  $x$  gibt so dass  $f(y) \leq f(x)$  für alle  $y \in U \cap A$ .

**Lemma 10.3.** Sei  $A \subseteq \mathbb{R}^n$  konvex,  $f : A \rightarrow \mathbb{R}$  linear und  $x$  ein lokales Optimum von  $f$ , dann ist  $x$  auch ein globales Optimum.

*Beweis.* Sei  $U$  eine Umgebung von  $x$  so dass  $f(y) \leq f(x)$  für alle  $y \in U \cap A$ . Sei  $z \in A$ . Dann existiert aufgrund der Konvexität von  $A$  ein  $t \in (0, 1)$  so dass  $tx + (1-t)z \in U \cap A$ . Damit ist  $f(x) \geq f(tx + (1-t)z)$  und, aufgrund der Linearität von  $f$ ,  $f(x) \geq tf(x) + (1-t)f(z)$ . Also  $f(x) \geq f(z)$ .  $\square$

**Lemma 10.4.** Falls  $L = F, S, A, b, c, \nu$  eine Schlupfform mit zulässiger Basislösung ist und die Simplex-Schleife, angewandt auf  $L$ , terminiert dann gilt:

1. Falls  $L$  unbeschränkt ist, dann antwortet die Simplex-Schleife mit “unbeschränkt”.
2. Falls  $L$  eine optimale Lösung mit Zielwert  $z \in \mathbb{R}$  hat, dann liefert die Simplex-Schleife eine Lösung mit Zielwert  $z$ .

*Beweis.* Die Simplex-Schleife kann nur entweder mit “unbeschränkt” oder mit einer Lösung antworten. Wir beginnen mit den folgenden beiden Beobachtungen: 1. Falls die Simplex-Schleife mit einer Lösung  $x$  antwortet, dann ist  $x$  ein lokales Optimum weil alle  $c_i \leq 0$  sind und damit wegen Lemma 10.3 auch ein globales Optimum. 2. Falls die Simplex-Schleife mit “unbeschränkt” antwortet, dann ist  $(x_1, \dots, x_{f-1}, a, x_{f+1}, \dots, x_{|S|+|F|})$  für alle  $a \geq x_f$  zulässig. Da  $c_f > 0$  ist damit  $L$  unbeschränkt.

Falls also  $L$  unbeschränkt ist, dann kann wegen 1. die Simplex-Schleife nicht mit einer Lösung antworten (sonst hätte  $L$  ja eine optimale Lösung), also antwortet  $L$  mit “unbeschränkt”. Falls  $L$  eine optimale Lösung hat, dann kann wegen 2. die Simplex-Schleife nicht mit “unbeschränkt” antworten (sonst wäre  $L$  ja unbeschränkt), also antwortet  $L$  mit einer Lösung, die nach 1., eine optimale Lösung ist.  $\square$

Wir wenden uns nun der Initialisierung zu. Zunächst kann man leicht beobachten, dass die Basislösung einer Schlupfform zulässig ist genau dann wenn  $b \geq 0$  ist. Für die Initialisierung des Simplex-Verfahrens (Phase 1) ist also vor allem der Fall wo  $b \not\geq 0$  ist interessant. In diesem Fall gehen wir wie folgt vor:

**Definition 10.10.** Sei  $L = A, b, c$  ein lineares Programm in Standardform mit  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^m$  und  $c \in \mathbb{R}^n$  so dass  $b \not\geq 0$ . Sei  $k \in \{1, \dots, m\}$  so dass  $b_k$  minimal ist. Dann definieren wir das lineare Programm  $S_L$  in Schlupfform in den Variablen  $\{x_0, \dots, x_{m+n}\}$  mit  $F = \{1, \dots, n, n+k\}$  und  $S = \{0, n+1, \dots, n+k-1, n+k+1, \dots, n+m\}$  als:

$$z = b_k - \sum_{j=1}^n a_{k,j} x_j - x_{n+k} \max!$$

$$x_0 = -b_k + \sum_{j=1}^n a_{k,j} x_j + x_{n+k}$$

$$x_{n+i} = (b_i - b_k) + \sum_{j=1}^n (a_{k,j} - a_{i,j}) x_j + x_{n+k} \quad \text{für } i \in \{1, \dots, k-1, k+1, \dots, m\}$$

**Lemma 10.5.** Sei  $L = A, b, c$  ein lineares Programm mit  $b \not\geq 0$ , dann ist die Basislösung von  $S_L$  zulässig. Weiters ist  $L$  lösbar genau dann wenn  $S_L$  eine optimale Lösung hat mit  $x_0 = 0$ .

*Beweis.* Zunächst beobachten wir dass die Basislösung von  $S_L$  zulässig ist, da  $-b_k \geq 0$  und, für alle  $i \in \{1, \dots, m\}$  gilt dass  $b_i \geq b_k$  und damit  $b_i - b_k \geq 0$ .

Um die zweite Behauptung zu zeigen, transformieren wir zunächst  $L =$

$$z = \sum_{j=1}^n c_j x_j \max!$$

$$\sum_{j=1}^n a_{i,j} x_j \leq b_i \quad \text{für } i = 1, \dots, m$$

$$x_j \geq 0 \quad \text{für } j = 1, \dots, n$$

in das folgende Hilfsprogramm  $L'$  in Standardform:

$$z = -x_0 \text{ max!}$$

$$\sum_{j=1}^n a_{i,j}x_j - x_0 \leq b_i \quad \text{für } i = 1, \dots, m$$

$$x_j \geq 0 \quad \text{für } j = 0, \dots, n$$

Zunächst sieht man dass  $L'$  lösbar ist: für beliebige  $x_1, \dots, x_n \geq 0$  wählen wir einfach  $x_0$  hinreichend groß um alle Ungleichungen zu erfüllen. Weiters gilt:  $L$  ist lösbar genau dann wenn  $L'$  eine optimale Lösung mit  $x_0 = 0$  hat. Falls nämlich  $L$  lösbar ist, dann erhält man durch Setzung von  $x_0 = 0$  eine Lösung von  $L'$  und diese Lösung ist optimal da ja  $x_0 \geq 0$  sein muss. Für die Gegenrichtung sei  $(0, x_1, \dots, x_n)$  optimale Lösung von  $L'$ , dann ist  $(x_1, \dots, x_n)$  auch Lösung von  $L$ .

Die Schlupfform von  $L'$  ist:

$$z = -x_0 \text{ max!}$$

$$x_{n+i} = b_i - \sum_{j=1}^n a_{i,j}x_j + x_0 \quad \text{für } i = 1, \dots, m$$

$$x_j \geq 0 \quad \text{für } j = 0, \dots, n + m$$

mit  $F = \{0, 1, \dots, n\}$  und  $S = \{n + 1, \dots, n + m\}$ . Ein Austausch von  $x_0$  mit  $x_{n+k}$  führt dann über die Gleichung  $x_0 = -b_k + \sum_{j=1}^n a_{k,j}x_j + x_{n+k}$  zu  $S_L$ .  $\square$

Auf Basis dieses Resultats kann jetzt die Prozedur  $\text{INITSIMPLEX}(A, b, c)$  wie in Algorithmus 44 angegeben werden, wobei wir davon ausgehen dass die Prozedur  $\text{SIMPLEX-SCHLEIFE}$  mit einer Schlupfform als Ergebnis antwortet. Damit ist das Simplex-Verfahren vollständig spezifiziert

---

#### Algorithmus 44 Initialisierung des Simplex-Algorithmus

---

**Prozedur**  $\text{INITSIMPLEX}(A, b, c)$

**Falls**  $b \geq 0$  **dann**

**Antworte**  $\text{SCHLUPFFORM}(A, b, c)$

**sonst**

$U := \text{SIMPLEX-SCHLEIFE}(S_{A,b,c})$

$(x_0, \dots, x_{n+m}) := \text{BASISLÖSUNG}(U)$

**Falls**  $x_0 = 0$  **dann**

$U' := \text{Transformiere } U \text{ in Schlupfform von } A, b, c$

**Antworte**  $U'$

**sonst**

**Antworte** "unlösbar"

**Ende Falls**

**Ende Falls**

**Ende Prozedur**

---

und wir können nun seine partielle Korrektheit beweisen.

**Satz 10.1.** *Sei  $L = A, b, c$  ein lineares Programm in Standardform so dass  $\text{SIMPLEX}(A, b, c)$  terminiert, dann gilt:*

1. Falls  $L$  unbeschränkt ist, dann antwortet  $\text{SIMPLEX}(A, b, c)$  mit "unbeschränkt".

2. Falls  $L$  unlösbar ist, dann antwortet  $\text{SIMPLEX}(A, b, c)$  mit “unlösbar”.

3. Falls  $L$  eine optimale Lösung mit Zielwert  $z \in \mathbb{R}$  hat, dann liefert  $\text{SIMPLEX}(A, b, c)$  eine Lösung mit Zielwert  $z$ .

*Beweis.* Da  $\text{SIMPLEX}(A, b, c)$  terminiert, terminiert auch die Simplex-Schleife im Aufruf  $\text{INITSIMPLEX}(A, b, c)$  mit, nach Lemma 10.4, der optimalen Lösung  $x = (x_0, \dots, x_{n+m})$  von  $S_L$ . Falls  $L$  unlösbar ist, dann ist nach Lemma 10.5  $x_0 \neq 0$  und damit antwortet die Prozedur  $\text{INITSIMPLEX}(A, b, c)$  mit “unlösbar” und das Verfahren wird abgebrochen. Falls  $L$  lösbar ist, dann ist nach Lemma 10.5  $x_0 = 0$  und damit erzeugt die Prozedur  $\text{INITSIMPLEX}(A, b, c)$  eine Schlupfform von  $A, b, c$  mit zulässiger Basislösung. Da  $\text{SIMPLEX}(A, b, c)$  terminiert, terminiert nun auch die Simplex-Schleife und damit folgt dieser Satz unmittelbar aus Lemma 10.4.  $\square$

Offen ist jetzt noch die Termination des Simplex-Verfahrens. Dazu wollen wir uns zunächst überlegen wie es passieren kann, dass der Zielwert nicht strikt steigt. Da nur solche  $f \in F$  ausgewählt werden die  $c_f > 0$  haben kann dieser Fall nur eintreten wenn eine Gleichung für ein  $s \in S$  gemeinsam mit ihrer Nichtnegativitätsbedingung  $x_s \geq 0$  die Variable  $x_f$  so einschränkt dass  $x_f$  gar nicht erhöht werden kann. In diesem Fall muss  $b_s = 0$  sein und wir sprechen von einer degenerierten Schlupfform.

*Beispiel 10.7.* Betrachten wir zum Beispiel

$$\begin{aligned}z &= 3 - \frac{1}{2}x_1 + 2x_2 \\x_3 &= 1 - \frac{1}{2}x_1 \\x_4 &= x_1 - x_2\end{aligned}$$

Diese Schlupfform hat die zulässige Basislösung  $(0, 0, 1, 0)$  mit Zielwert 3. Dann wird die freie Variable  $x_2$  zum Vertauschen ausgewählt. Nun schränkt aber die Gleichung und Nichtnegativitätsbedingung von  $x_4$  die Variable  $x_2$  auf  $x_2 \leq 0$  ein. Wir erhalten durch den Austausch die Schlupfform

$$\begin{aligned}z &= 3 + \frac{3}{2}x_1 - 2x_4 \\x_2 &= x_1 - x_4 \\x_3 &= 1 - \frac{1}{2}x_1\end{aligned}$$

die ebenfalls die zulässige Basislösung  $(0, 0, 1, 0)$  mit Zielwert 3 hat. In diesem Schritt hat sich der Zielwert also nicht verändert. Nun muss die freie Variable  $x_1$  ausgewählt werden die durch die Gleichung von  $x_3$  am stärksten eingeschränkt wird. Nach Durchführung dieser Vertauschung erhalten wir die Schlupfform

$$\begin{aligned}z &= 6 - 3x_3 - 2x_4 \\x_1 &= 2 - 2x_3 \\x_2 &= 2 - 2x_3 - x_4\end{aligned}$$

mit Basislösung  $(2, 2, 0, 0)$  und Zielwert 6 wobei es sich, da alle  $c_j \leq 0$  sind, um eine optimale Lösung handelt.

Falls Zwischenschritte auftreten, in denen sich der Wert der Zielfunktion nicht verändert, spricht man von *stalling*. Das tritt in der Praxis häufig auf und ist (bis auf den Zeitaspekt) kein Problem solange der Algorithmus zu einem späteren Zeitpunkt wieder einen Schritt durchführt der den Zielwert strikt erhöht. Allerdings kann es auch dazu kommen, dass der Algorithmus kreist, d.h. dass er von einer Schlupfform  $U$  nach einigen Schritten die den Zielwert nicht erhöhen wieder bei  $U$  ankommt. Dieses Verhalten, wenn auch theoretisch möglich, tritt in der Praxis so selten auf, dass die meisten Implementierungen es gar nicht behandeln.

*Beispiel 10.8.* Die Schlupfform

$$\begin{aligned} z &= 10x_1 - 57x_2 - 9x_3 - 24x_4 \\ x_5 &= -\frac{1}{2}x_1 + \frac{11}{2}x_2 + \frac{5}{2}x_3 - 9x_4 \\ x_6 &= -\frac{1}{2}x_1 + \frac{3}{2}x_2 + \frac{1}{2}x_3 - x_4 \\ x_7 &= 1 - x_1 \end{aligned}$$

kann mit den Austauschschritten  $x_1 \leftrightarrow x_5$ ,  $x_2 \leftrightarrow x_6$ ,  $x_3 \leftrightarrow x_1$ ,  $x_4 \leftrightarrow x_2$ ,  $x_5 \leftrightarrow x_3$ ,  $x_4 \leftrightarrow x_6$  wieder auf sich selbst zurückgeführt werden.

Es gibt verschiedene Strategien um Kreisen zu verhindern, z.B. die Regel von Bland die darin besteht bei mehreren Möglichkeiten für die Auswahl eines  $x_i$  immer jene zu bevorzugen wo  $i$  minimal ist. Mit einer derartigen Erweiterung kann die Termination des Simplex-Verfahren sichergestellt werden. Für dieses Verfahren gilt dann eine entsprechend stärkere Form von Satz [10.1](#).



# Problemverzeichnis

1	Bestimmung des ggT . . . . .	1
2	Sortierproblem . . . . .	1
3	Linearisierung . . . . .	1
4	Arithmetisches Mittel . . . . .	3
5	Kürzester Pfad . . . . .	17
6	Problem des Handelsreisenden (engl. <i>Travelling Salesman Problem (TSP)</i> ) . . . . .	17
7	Knotenfärbung . . . . .	17
8	Minimaler Spannbaum . . . . .	21
9	Matrixmultiplikation . . . . .	25
10	Dichtestes Punktepaar . . . . .	29
11	Wörterbuchproblem . . . . .	49
12	Linearisierung (Topologisches Sortieren) . . . . .	72
13	Basis minimalen Gewichts . . . . .	82
14	Optimale Knotenüberdeckung . . . . .	83
15	Knotenüberdeckung . . . . .	85
16	Anzahl der Pfade . . . . .	87
17	Stabzerlegungsproblem . . . . .	89
18	Segmentierte einfache lineare Regression . . . . .	90
19	Bewerberproblem . . . . .	93
20	Primzahlen . . . . .	101
21	Lineare Optimierung . . . . .	107
22	Maximaler Fluss . . . . .	108
23	Schaltkreisevaluierung . . . . .	110
24	Lineare Optimierung (Standardform) . . . . .	111



# Algorithmenverzeichnis

1	Prozedur P . . . . .	3
2	Einfügesortieren . . . . .	5
3	Verschmelzen . . . . .	24
4	Sortieren durch Verschmelzen (engl. <i>merge sort</i> ) . . . . .	24
5	Matrixmultiplikation (direkt) . . . . .	26
6	Matrixmultiplikation (rekursiv) . . . . .	27
7	Strassen-Algorithmus . . . . .	28
8	Erschöpfende Suche nach dichtestem Punktepaar . . . . .	29
9	Auswahl aus einem Datenfeld . . . . .	31
10	Teile-und-herrsche Algorithmus für dichtestes Punktepaar . . . . .	32
11	Binäre Suche . . . . .	50
12	Suche in einem Suchbaum . . . . .	52
13	Durchlaufen eines Suchbaums in symmetrischer Reihenfolge . . . . .	52
14	Einfügen in einen Suchbaum . . . . .	52
15	Löschen aus einem Suchbaum . . . . .	54
16	Löschen der Wurzel aus einem Suchbaum . . . . .	55
17	Finde und lösche Minimum aus Suchbaum . . . . .	55
18	Stapel . . . . .	62
19	Warteschlange . . . . .	63
20	Einfügen in eine Prioritätswarteschlange . . . . .	65
21	Erhöhung der Priorität in einer Prioritätswarteschlange . . . . .	65
22	Extraktion des Maximums aus Prioritätswarteschlange . . . . .	66
23	Haldensortieren (engl. <i>heapsort</i> ) . . . . .	67
24	Breitensuche (engl. <i>breadth-first search</i> ) . . . . .	70
25	Tiefensuche (engl. <i>depth-first search</i> ) . . . . .	70
26	Linearisieren (Topologisches Sortieren) . . . . .	73
27	Algorithmus von Kruskal . . . . .	77
28	Algorithmus von Prim . . . . .	79
29	Algorithmus von Dijkstra . . . . .	80
30	Generischer gieriger Algorithmus . . . . .	82
31	Gierige Knotenüberdeckung . . . . .	84
32	Anzahl der Pfade (exponentiell) . . . . .	87
33	Anzahl der Pfade (top-down mit Memoisierung) . . . . .	88
34	Stabzerlegung (bottom-up) . . . . .	90
35	Segmentierte Methode der kleinsten Quadrate . . . . .	92
36	Direkte Lösung des Bewerberproblems . . . . .	94
37	Randomisierte Lösung des Bewerberproblems . . . . .	94
38	Der Fisher-Yates Algorithmus . . . . .	95
39	Quicksort . . . . .	96

40	Randomisiertes Quicksort . . . . .	98
41	Zählsortieren . . . . .	101
42	Miller-Rabin Primzahltest . . . . .	103
43	Der Simplex-Algorithmus . . . . .	115
44	Initialisierung des Simplex-Algorithmus . . . . .	117

# Literaturverzeichnis

- [1] Johannes A. Buchmann. *Einführung in die Kryptographie, 6. Auflage*. Springer, 2016.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [3] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.
- [4] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.
- [5] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.
- [6] Markus Nebel. *Entwurf und Analyse von Algorithmen*. Springer Vieweg, 2012.
- [7] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage*. Spektrum Akademischer Verlag, 2012.
- [8] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2012.
- [9] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 4th edition, 2014.