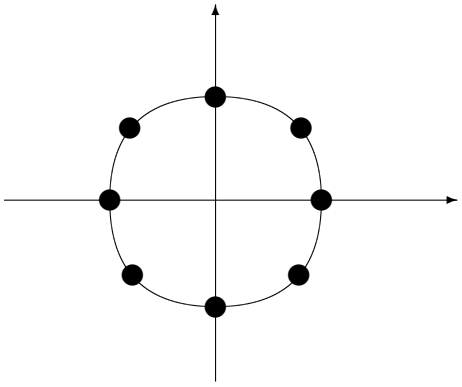
Skript zu
Algorithmen und Datenstrukturen

Uwe Schöning



Mitschrift und LaTeX-Bearbeitung:

Michael Henn

Inhaltsverzeichnis

Kapitel 1

Einige Analysetools

1.1 Asymptotische Notationen

Die genaue Angabe einer Komplexitätsfunktion, bis auf den konstanten Faktor, ist oft schwierig oder unmöglich, da die konkrete Laufzeit eines Algorithmus von der verwendeten Maschine, der Programmiersprache, ja sogar vom verwendeten Compiler, abhängig ist. Eine Angabe, die dagegen stabil, also von der konkreten Ausführung des Algorithmus unabhängig ist, ist zum Beispiel die Aussage, dass die Komplexität „höchstens quadratisch mit der Eingabelänge zunimmt“. Genau solche unscharfen Aussagen werden durch die folgenden Notationen unterstützt.

Wir beginnen mit der $\mathcal{O}$-Notation, welche es ermöglicht, Aussagen über obere Schranken zu machen – unter Ignorieren von konstanten Faktoren und von Termen niedrigerer Ordnung.

Im Folgenden seien f und g immer Funktionen von $\mathbb{N}$ nach $\mathbb{R}_+$.

DEFINITION ($\mathcal{O}$-Notation):
Mit $\mathcal{O}(f(n))$ bezeichnen wir die Klasse aller Funktionen g mit der Eigenschaft:

$$\exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$

Beispiel:
Es gilt $3n^4 + 5n^3 + 7\log n \in \mathcal{O}(n^4)$, denn es gilt $3n^4 + 5n^3 + 7\log n \leq 3n^4 + 5n^4 + 7n^4 = 15n^4$ für $n \geq 1$. Daher können wir $c = 15$ und $n_0 = 1$ wählen.

Die $\mathcal{O}$-Notation ist natürlich nur dann sinnvoll, wenn die zur Abschätzung verwendete Funktion *möglichst einfach* ist, also keine überflüssigen konstanten Faktoren oder Terme niedriger Ordnung enthält. Übliche Funktionen, die die Rolle von f übernehmen, sind $f(n) = n,\ , n \cdot \log n,\ n^2,\ n^3, \ldots, 2^n,\ n!$.

Wenn wir im Rahmen der $\mathcal{O}$-Notation den Logarithmus verwenden, also $\mathcal{O}(\ldots \log n \ldots)$, so erübrigt sich die Angabe der Basis des Logarithmus, da sich verschiedene Basen lediglich im konstanten Faktor unterscheiden. Konstante Faktoren werden durch die $\mathcal{O}$-Notation aber gerade ignoriert.

Dual zur $\mathcal{O}$-Notation, und damit für untere Schranken geeignet, ist die Ω-Notation:

DEFINITION (Ω-Notation):
Mit $\Omega(f(n))$ bezeichnen wir die Klasse aller Funktionen g mit der Eigenschaft:

$$\exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \ g(n) \geq c \cdot f(n)$$

DEFINITION (Θ-Notation):
Mit $\Theta(f(n))$ bezeichnen wir die Klasse $\mathcal{O}(f(n)) \cap \Omega(f(n))$.

Falls also $g(n) \in \Theta(f(n))$ gezeigt ist, so haben wir das Wachstum der Funktion $g(n)$ mittels $f(n)$ genau – bis auf einen konstanten Faktor und bis auf Terme niedriger Ordnung – abgeschätzt. Die Funktion $g(n)$ verläuft dann also, zumindest ab einem Anfangswert n_0, in dem „Streifen" $[c_1 f(n), c_2 f(n)]$, für gewisse Konstanten c_1, c_2 mit $c_1 < c_2$.

Beispiel:
$1 + \sin n \in \Theta(1),\ 3n^2 \log n + 4n^2 + 3n \in \Theta(n^2 \log n)$.

Zum Schluss wollen wir noch eine weitere asymptotische Notation einführen: Man schreibt $f(n) \sim g(n)$ genau dann, wenn $\lim_{n\to\infty} f(n)/g(n) = 1$. Dies ist ein genauerer Vergleich der beiden Funktionen f und g als dies durch die Notation $f(n) = \Theta(g(n))$ ausgedrückt wird. Bei $f(n) \sim g(n)$ muss der konstante Faktor bei beiden Funktionen übereinstimmen.

1.2 Worst-Case und Average-Case

Ein *Algorithmus* wird immer gestartet mit einer *Eingabe*. Eine Eingabe kann zum Beispiel ein Graph, eine Zahl, eine Matrix, etc., oder eine Kombination hiervon sein. Jeder Eingabe kann man eine *Länge* $n \in \mathbb{N}$ zuordnen. Mit $|x|$ bezeichnen wir die Länge der Eingabe x.

Beispiel:

- Graph: Anzahl Knoten = n; Anzahl Kanten = m; $|G| = n + m$
- Zahl $x = (x_{n-1}x_{n-2}\ldots x_0)_2$: Länge n
- Folge $a_1, \ldots, a_n$: Länge n

Im Allgemeinen gehen wir davon aus, dass eine einfache arithmetische Operation $x = y + z$ oder ein reines Umspeichern $x = y$ mit einer bzw. $\mathcal{O}(1)$ Rechenzeit-Einheit veranschlagt werden kann (uniformes Kostenmaß). Dies ist dann gerechtfertigt, wenn die beteiligten Zahlen nicht zu groß werden und in eine Speicherzelle bzw. ein Register passen. Sollten Zahlen allerdings sehr groß werden, so ist das logarithmische Kostenmaß besser angebracht, das das Umspeichern einer Zahl x mit $\log x$ Rechenzeiteinheiten

veranschlägt. Dies entspricht der Anzahl der Bits, die umgespeichert bzw. verarbeitet werden (auch *Bit-Komplexität* genannt). Insbesondere bei Algorithmen, die Zahlen als Eingabe verarbeiten (siehe Kapitel 7), ist die Bit-Komplexität anzuwenden. Das bedeutet, dass eine Zahl x als Eingabe, die n Bits in Anspruch nimmt (also $\log x \approx n$) mit der Eingabelänge n zu berücksichtigen ist. Im Sinne der Bit-Komplexität hat damit die Addition zweier n-Bitzahlen den Aufwand $\mathcal{O}(n)$ und die Multiplikation bzw. Division den Aufwand $\mathcal{O}(n^2)$

DEFINITION (worst-case-Komplexität):
Die worst-case-Komplexität *von einem Algorithmus A ist die Laufzeit von A bei der ungünstigsten Eingabe der Länge n. In Formeln:*

$$\textit{wc-time}_A(n) \;:=\; \max_{x:|x|=n} \textit{time}_A(x)$$

Hierbei ist $\textit{time}_A(x)$ die Laufzeit (Anzahl der durchlaufenen Elementarschritte) von Algorithmus A bei Eingabe x.

Da wir an dieser Stelle nur über *deterministische* Algorithmen reden, ist mit Angabe einer Eingabe x der Rechenablauf und damit dessen Länge eindeutig festgelegt.

Falls wc-time$_A(n) \le f(n)$ für eine Funktion f gilt, so heißt dies, dass der Algorithmus A bei *jeder* Eingabe der Länge n höchstens $f(n)$ Schritte macht. Falls dagegen wc-time$_A(n) \ge g(n)$ für eine Funktion g gilt, so heißt dies (nur), dass es Eingaben der Länge n gibt, für die Algorithmus A mindestens die Laufzeit $g(n)$ benötigt.

DEFINITION (average-case-Komplexität):
Bei der average-case-*Analyse wird Gleichverteilung auf allen Eingaben der Länge n angenommen. Die Laufzeit von einem Algorithmus A (bei zufälliger Wahl der Eingabe) wird damit zu einer* Zufallsvariablen $T_{A,n}$. *Es ist*

$$\textit{av-time}_A(n) \;:=\; \mathbb{E}[\,T_{A,n}\,]$$

wobei die Eingaben der Länge n unter Gleichverteilung zufällig ausgewählt werden. Hierbei ist $\mathbb{E}$ der Erwartungswertoperator. *In Formeln heißt dies also:*

$$\textit{av-time}_A(n) \;:=\; \frac{1}{|\{x : |x| = n\}|} \cdot \sum_{x:|x|=n} \textit{time}_A(x)$$

Es ist klar, dass immer av-time$_A(n) \le$ wc-time$_A(n)$ gilt.

Eine average-case Analyse ergibt eher realistischere Aussagen über die Performanz eines Algorithmus als eine worst-case Analyse; ist jedoch im allgemeinen schwieriger durchzuführen. Das Unrealistische wiederum an einer average-case Analyse ist die Annahme der Gleichverteilung. In dieser Hinsicht ist die worst-case Analyse wieder robuster, da sie keine spezielle Annahme über die Verteilung der Eingaben macht.

Beispiel (Maximum bestimmen):
Betrachte den folgenden Programmabschnitt

Algorithmus 1.1 `findMax`$(a[1..n])$

Voraussetzung: $n \geq 1$

```
1: max := a[1]
2: FOR i := 2 TO n DO
3:    IF a[i] > max THEN
4:       max := a[i]
```

Sei c_1 der Aufwand zur Ausführung von (1). Sei c_2 der Aufwand, der mit jedem einzelnen Schleifendurchlauf in (2) und (3) verbunden ist. Sei c_3 der Aufwand für (4).
Dann ergibt sich:

$$\text{wc-time}_A(n) \;=\; c_1 + (n-1)(c_2 + c_3)$$

Wie sieht es mit dem average-case aus? Als eine Eingabe der Länge n betrachten wir hier ein Array $a[1..n]$, dessen Elemente eine beliebige Permutation der Zahlen (z.B.) $1, 2, \ldots, n$ darstellen. Alle $n!$ solchen Permutationen seien gleichwahrscheinlich. Dann gilt:

$$\begin{aligned} Pr(a[2] > a[1]) &= 1/2 \\ Pr(a[3] > \max(a[1], a[2])) &= 1/3 \\ &\vdots \\ Pr(a[n] > \max(a[1], a[2], \ldots, a[n-1])) &= 1/n \end{aligned}$$

Sei X_j $(j = 1, \ldots, n)$ die Zufallsvariable mit

$$X_j = \begin{cases} 1, & \text{in dem Arrayabschnitt } a[1..j] \text{ ist } a[j] \text{ das Maximum} \\ 0, & \text{sonst} \end{cases}$$

Dann ist $\mathbb{E}[X_j] = Pr(a[j] > \max(a[1], a[2], \ldots, a[j-1])) = 1/j$. Daher ergibt sich:

$$\begin{aligned} \text{av-time}_A(n) &= c_1 + (n-1)c_2 + c_3 \cdot \mathbb{E}\left[\sum_{j=2}^{n} X_j\right] \\ &= c_1 + (n-1)c_2 + c_3 \sum_{j=2}^{n} \mathbb{E}[X_j] \\ &= c_1 + (n-1)c_2 + c_3 \sum_{j=2}^{n} \frac{1}{j} \\ &= c_1 + (n-1)c_2 + (H_n - 1)c_3 \\ &\approx c_1 + (n-1)c_2 + (\ln n)c_3 \end{aligned}$$

Hierbei ist $H_n = \sum_{j=1}^{n} \frac{1}{j}$ die *harmonische Reihe*. Wir verwenden hier (und später) die Abschätzung

$$H_n \approx \ln n$$

Zu beachten ist, dass die Analyse stochastischer Algorithmen von der average-case Analyse deterministischer Algorithmen unterschieden werden muss. Bei probabilistischen Algorithmen A muss $\text{time}_A(x)$ als Zufallsvariable betrachtet werden:

- $\text{wc-time}_A(n) = \max\{\overbrace{\mathbb{E}}^{\text{Zufall innerhalb } A}(\text{time}_A(x)) \mid |x| = n\}$
- $\text{av-time}_A(n) = \underbrace{\mathbb{E}}_{x:|x|=n}(\overbrace{\mathbb{E}}^{\text{Zufall innerhalb } A}(\text{time}_A(x)))$

1.3 Einige nützliche Abschätzungen

Die folgenden Funktionen treten oftmals bei der Analyse von Algorithmen auf. Oft lassen sich diese durch einfachere Funktionen (asymptotisch) abschätzen.

Es gilt mit der *Stirlingschen Formel*

$$n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

Hieraus ergibt sich

$$\log(n!) = n \log n - \Theta(n)$$

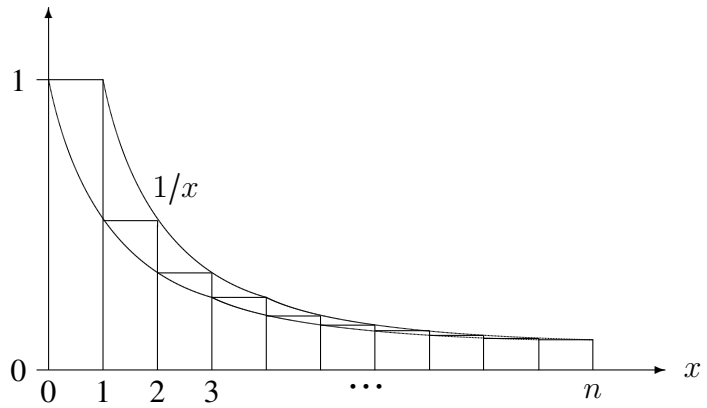
Des Weiteren sei

$$H_n := \sum_{i=1}^{n} \frac{1}{i}$$

die *harmonische Reihe*. Es gilt:

$$\ln(n+1) \leq H_n \leq \ln n + 1$$

wobei $\ln$ der natürliche Logarithmus ist. Diese Abschätzung lässt sich einsehen, indem man die Summenfunktion H_n von oben und von unten durch ein Integral abschätzt: $\ln(n+1) = \int_1^{n+1} (1/x)dx \leq H_n \leq 1 + \int_1^n (1/x)dx = 1 + \ln n$, siehe folgende Skizze:

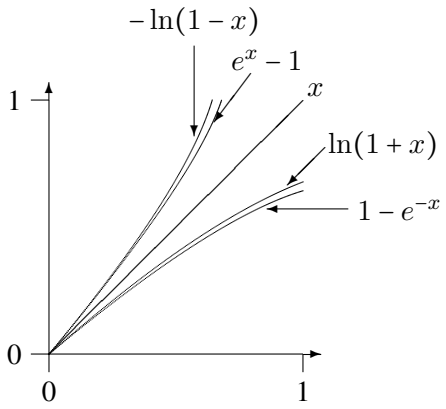


Tatsächlich gilt die folgende, etwas genauere Abschätzung:

$$H_n \sim \ln n + \frac{1}{2n} + \gamma$$

wobei $\gamma \approx 0.57721$ (die Eulersche Konstante).

Gelegentlich möchte man Ausdrücke mit Logarithmen vermeiden und einen linearen Ausdruck verwenden; oder aber man möchte mit einer Exponentialfunktion abschätzen. Das folgende Diagramm gibt eine Reihe von unteren und oberen Abschätzungen der Funktionen $f(x) = x$ an, die besonders gut in der Nähe von $x = 0$ sind.



Diese Abschätzungen lassen sich einsehen, indem man die Potenzreihen der entsprechenden Exponential- und Logarithmusfunktionen inspiziert. Es gilt:

$$\begin{aligned} e^x - 1 &= x + \tfrac{x^2}{2} + \tfrac{x^3}{6} + \tfrac{x^4}{24} + \cdots \\ -\ln(1-x) &= x + \tfrac{x^2}{2} + \tfrac{x^3}{3} + \tfrac{x^4}{4} + \cdots \\ \ln(1+x) &= x - \tfrac{x^2}{2} + \tfrac{x^3}{3} - \tfrac{x^4}{4} + -\cdots \\ 1 - e^{-x} &= x - \tfrac{x^2}{2} + \tfrac{x^3}{6} - \tfrac{x^4}{24} + -\cdots \end{aligned}$$

Hieraus lassen sich leicht weitere Abschätzungen gewinnen. Beispielsweise gilt $1 + x \le e^x$ und $1 - x \le 1 - \ln(1+x) \le e^{-x}$, usw.

Nützliche Abschätzungen ergeben sich auch aus der Beziehung zwischen verschiedenen Mittelwertbildungen. Seien $a_i > 0$, $i = 1, \ldots, n$. Dann gilt folgende Abschätzung zwischen dem *harmonischen Mittel*, dem *geometrischen Mittel*, dem *arithmetischen Mittel* und dem *quadratischen Mittel* (Satz von Cauchy):

$$\begin{aligned} \min(a_1, \ldots, a_n) &\le \frac{n}{1/a_1 + 1/a_2 + \cdots + 1/a_n} && \text{(harmon. Mittel)} \\ &\le (a_1 \cdot a_2 \cdots a_n)^{1/n} && \text{(geomet. Mittel)} \\ &\le (a_1 + a_2 + \cdots + a_n)/n && \text{(arithm. Mittel)} \\ &\le \sqrt{(a_1^2 + \cdots + a_n^2)/n} && \text{(quadrat. Mittel)} \\ &\le \max(a_1, \ldots, a_n) \end{aligned}$$

Aus der Beziehung zwischen dem geometrischen und dem arithmetischen Mittel ergibt sich nach Logarithmieren

$$\frac{1}{n}\sum_{i=1}^{n}\ln a_i \;=\; \frac{1}{n}\ln\left(\prod_{i=1}^{n}a_i\right) \;=\; \ln\left((\prod_{i=1}^{n}a_i)^{1/n}\right) \;\leq\; \ln\left(\frac{1}{n}\sum_{i=1}^{n}a_i\right)$$

Dies kann als Abschätzung zwischen zwei Erwartungswerten umgeschrieben werden:

$$\mathbb{E}[\ln X] \leq \ln(\mathbb{E}[X]) \;\text{ oder }\; \mathbb{E}[Y] \leq \ln(\mathbb{E}[e^Y]) \;\text{ oder }\; \mathbb{E}[Z] \leq \log_2(\mathbb{E}[2^Z])$$

Dies nennt man die *Jensensche Ungleichung*.
Hierbei wurde die folgende Verallgemeinerung der Beziehung zwischen geometrischem und arithmetischem Mittel verwendet. Sei $p_1, \ldots, p_n$ mit $\sum_{i=1}^{n} p_i = 1$ eine Wahrscheinlichkeitsverteilung. Dann gilt:

$$\prod_{i=1}^{n} a_i^{p_i} \;\leq\; \sum_{i=1}^{n} p_i a_i$$

Sei $\pi(n)$ die Anzahl der Primzahlen bis zur Zahl n. Der berühmte *Primzahlsatz* der Zahlentheorie besagt, dass

$$\pi(n) \;\sim\; n/\ln n$$

Wenn man also eine Zahl $z \in \{1, \ldots, n\}$ zufällig unter Gleichverteilung zieht, so wird diese etwa mit Wahrscheinlichkeit $\frac{n/\ln n}{n} = 1/\ln n$ eine Primzahl sein. Im Mittel wird man also das Zufallsexperiment $(\ln n)$-mal wiederholen müssen, bis man eine Primzahl gefunden hat.

1.4 Rekursionsgleichungen

Bei der Komplexitätsanalyse von (rekursiven) Algorithmen entstehen sehr oft Rekursionsgleichungen, die behandelt und gelöst werden müssen. Eine Rekursionsgleichung hat die allgemeine Form $f(n) = F(f(1), f(2), \ldots, f(n-1))$.
Ohne eine weitere Angabe besteht die Lösungsmenge einer solchen Gleichung in einer Funktionenschar. Die einzelnen Funktionen der Funktionenschar werden durch Festlegen geeigneter Parameter spezifiziert. Wenn k Parameter anzugeben sind, so können diese durch Lösen von k Anfangswertbedingungen an f ermittelt werden:

$$\begin{aligned} f(1) &= a_1 \\ f(2) &= a_2 \\ &\vdots \\ f(k) &= a_k \end{aligned}$$

Beispiel:
Die *Fibonacci*-Folge ist definiert durch die Rekursion $f(n) = f(n-1) + f(n-2)$ und $f(1) = f(2) = 1$. Wir vermuten, dass diese Funktion exponentielles Wachstum besitzt.

Also zeigen wir, dass für gewisse Konstanten a, c gilt: $f(n) \geq ac^n$, $n \geq n_0$. Wir setzen induktiv in die Rekursionsgleichung ein:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &\geq ac^{n-1} + ac^{n-2} \\ &= ac^n \cdot \frac{c+1}{c^2} \end{aligned}$$

Dieser letzte Ausdruck sollte $\geq ac^n$ sein, damit der Induktionsschritt korrekt bewiesen ist. Dies ist äquivalent mit $c^2 - c - 1 \leq 0$. Lösen der quadratischen Gleichung $c^2 - c - 1 = 0$ liefert $c = \frac{\sqrt{5}+1}{2}$ (der goldene Schnitt). (Die zweite, negative Lösung für c ist hier uninteressant). Somit ist der Induktionsschritt für $c \leq \frac{\sqrt{5}+1}{2} \approx 1.618$ gezeigt.
Der Induktionsanfang gilt, indem man geeignet a und den Startwert n_0 wählt.
Analog (durch Vertauschen von „≥“ und „≤“) zeigt man auch, dass $f(n) \leq a'd^n$ für eine geeignete Konstante a' und $d \geq \frac{\sqrt{5}+1}{2}$ gilt.
Zusammengefasst heißt dies, dass $f(n) = \Theta(c^n)$, wobei $c = \frac{\sqrt{5}+1}{2}$.

1.5 Konstruktive Induktion

Beispiel:
Angenommen, ein typischer *divide-and-conquer* Algorithmus zerlegt das zu lösende Problem der Größe $n = 2^k$, $k \geq 1$, in 2 Teilprobleme der Größe $n/2$, löst diese durch rekursive Aufrufe und setzt die Lösung aus den beiden erhaltenen Lösungen zusammen:

Algorithmus 1.2 `loese(`x `: Eingabe) : Lösung`

```
1: IF |x| = 1 THEN
2:    löse x direkt; die Lösung sei l
3:    return l
4: ELSE
5:    zerlege x in zwei Teilprobleme x_1 und x_2
6:    l_1 := loese(x_1)
7:    l_2 := loese(x_2)
8:    setze Lösung l aus l_1 und l_2
9:    return l
```

Der Aufwand für das Zerlegen der Eingabe und Zusammenfügen der Lösungen zur Gesamtlösung sei durch $cn + d$ gegeben. Dann ergibt sich für die Komplexität von `loese` bei einer Eingabe der Länge n:

$$f(n) = cn + d + 2f(n/2)$$

Nach etwas Nachdenken wird man vielleicht eine Lösung der Form $f(n) = \alpha + \beta n + \delta n \log_2 n$ (mit noch unbekannten Konstanten α, β, δ) vermuten. Setzen wir dies (induk-

tiv) ein:

$$\begin{aligned} f(n) &= cn + d + 2f(n/2) \\ &= cn + d + 2\alpha + 2\beta(n/2) + 2\delta(n/2)\log_2(n/2) \\ &= cn + d + 2\alpha + \beta n + \delta n \log_2 n - \delta n \end{aligned}$$

Durch Koeffizientenvergleich mit der erwünschten Lösung $\alpha + \beta n + \delta n \log_2 n$ ergibt sich $\alpha = -d$ und $\delta = c$. Die Funktionenschar der Lösungsfunktionen hat also die Form

$$\{-d + \beta n + cn \log_2 n \mid \beta \in \mathbb{R}\}$$

Sei nun die Anfangswertbedingung $f(1) = a$ gegeben. Dann ergibt sich durch Einsetzen der Wert des Parameters β:

$$-d + \beta + c\log_2 1 = a \quad \Rightarrow \quad \beta = a + d$$

Die Lösungsfunktion lautet also $f(n) = -d + (a + d)n + cn \log_2 n$.

Dies war ein Beispiel für die *induktive Einsetzungsmethode*. Man rät die Lösung und bestätigt diese durch Induktion. Oftmals enthält die vermutete Lösung noch zu spezifizierende Parameter. Die Werte dieser Parameter ergeben sich dann oft durch die Notwendigkeit, dass der Induktionsbeweis korrekt vollzogen werden muss (vgl. den Koeffizientenvergleich oben). In diesem Zusammenhang spricht man auch oft von einer *konstruktiven Induktion*.

Bei dem folgenden *Beispiel* gibt es einen kleinen Haken. Gegeben sei die Rekursionsgleichung $f(n) = f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + 1$. Wir raten die durchaus richtige Lösung $f(n) = \mathcal{O}(n)$. Also versuchen wir es mit $f(n) \leq cn$ mittels der induktiven Einsetzungsmethode:

$$\begin{aligned} f(n) &= f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + 1 \\ &\leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 \\ &= cn + 1 \end{aligned}$$

So, und wie soll es nun weitergehen? Der letzte Ausdruck lässt sich nicht durch cn nach oben abschätzen.

Der Trick besteht darin, bei der Lösungsvermutung eine Konstante abzuziehen. Es soll also die *stärkere* Behauptung $f(n) \leq cn - a$ bewiesen werden.

$$\begin{aligned} f(n) &= f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + 1 \\ &\leq (c\lceil n/2 \rceil - a) + (c\lfloor n/2 \rfloor - a) + 1 \\ &= cn - 2a + 1 \\ &\leq cn - a \quad \text{falls } a \geq 1. \end{aligned}$$

Bei der *Iterationsmethode* wird fortgesetzt die Rekursionsgleichung auf der rechten Seite wieder eingesetzt und dann, wenn möglich, in eine geschlossene Form aufgelöst.

Beispiel ($f(n) = n + 3f(n/4)$):
Wir setzen iterativ ein:

$$\begin{aligned} f(n) &= n + 3f(\tfrac{n}{4}) = n + 3(\tfrac{n}{4} + 3f(\tfrac{n}{16})) = n + \tfrac{3}{4}n + 3^2 f(\tfrac{n}{4^2}) \\ &= n + \tfrac{3}{4}n + 3^2(\tfrac{n}{4^2} + 3f(\tfrac{n}{4^3})) = n + \tfrac{3}{4}n + (\tfrac{3}{4})^2 n + 3^3 f(\tfrac{n}{4^3}) \\ &= n + \tfrac{3}{4}n + (\tfrac{3}{4})^2 n + \cdots + (\tfrac{3}{4})^{\log_4 n - 1} + 3^{\log_4 n} f(1) \\ &\leq n \cdot \sum_{i=0}^{\infty} (\tfrac{3}{4})^i + 3^{\log_4 n} f(1) = n \cdot \tfrac{1}{1-3/4} + \mathcal{O}(n^c),\ c < 1 \\ &= 4n + \mathcal{O}(n^c) = \mathcal{O}(n) \end{aligned}$$

Da $f(n) \geq n$, gilt also $f(n) = \Theta(n)$. Man beachte, dass man diese asymptotische Aussage treffen kann, *ohne* eine Anfangswertbedingung verwendet zu haben, denn die Anfangswertbedingung $f(1) = a$ wirkt sich höchstens auf die in der Θ-Notation versteckte Konstante aus.

Die nächste Methode, die wir diskutieren wollen, ist die *Variablensubstitution*.
Einige Arten von Rekursionsgleichungen haben wir bereits im Griff und können sie also lösen. Andere scheinen auf den ersten Blick überhaupt nicht die vertraute Form zu haben:

$$f(n) = 2f(\sqrt{n}) + \log_2 n$$

Das Ungewohnte hierbei ist, dass das Argument von f auf der rechten Seite nicht die Form $n/2$ (oder $n - 1$) hat.
Führen wir aber mal eine neue Variable m ein und setzen $m = \log n$ bzw. $n = 2^m$. Dann ergibt sich:

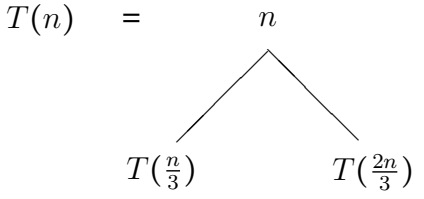
$$f(2^m) = 2f(\sqrt{2^m}) + \log_2(2^m) = 2f\left(2^{\frac{m}{2}}\right) + m$$

Kürzen wir ferner $f(2^m)$ durch $g(m)$ ab, so erhalten wir nun die neue Rekursionsgleichung:

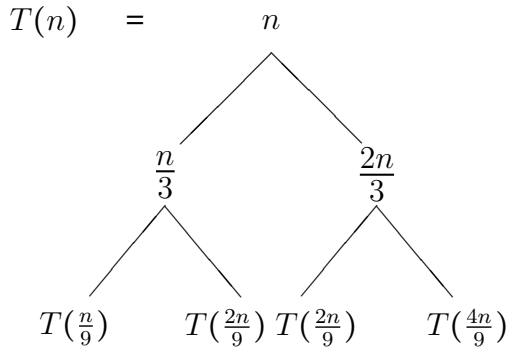
$$g(m) = 2g\left(\frac{m}{2}\right) + m$$

Diese Gleichung können wir lösen; siehe oben. Es ist $g(m) = \Theta(m \log m)$. Wieder rückwärts eingesetzt ergibt dies $f(2^m) = g(m) = \Theta(m \log m)$, und damit, wegen $m = \log_2 n$, $f(n) = \Theta(\log n \log\log n)$.

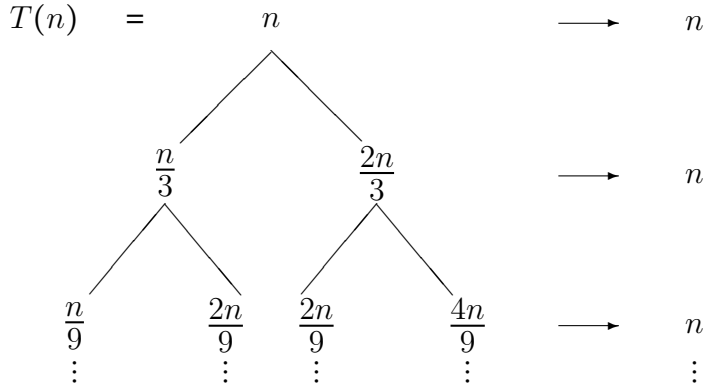
Ähnlich der iterativen Einsetzungsmethode ist die *Visualisierung der Baumstruktur:* Wir entwickeln die Rekursionsgleichung Schritt für Schritt in eine Baumstruktur:
Sei beispielsweise $T(n) = n + T(n/3) + T(2n/3)$ gegeben.



Und weiter:



Und weiter:



Man erkennt, dass in jeder Schicht des Baumes ein Anteil von n (in den tieferen Schichten $\leq n$) entsteht. Da die Baumtiefe durch $\mathcal{O}(\log n)$ beschränkt ist, ergibt sich $T(n) \leq n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.

Bei den obigen Beispielen sind bereits verschiedene Vereinfachungen und Vernachlässigungen vorgekommen, die bei den Rekursionsgleichungen wie sie bei der Algorithmenanalyse vorkommen, allerdings üblich und durchaus zulässig sind. Wir stellen diese nochmals zusammen:

- *Nicht-Ganzzahligkeit.*

  Korrekt sollte die Rekursionsgleichung für die Laufzeit $T(n)$ eines gewissen divide-and-conquer Algorithmus vielleicht lauten

  $$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$$

  Stattdessen ignorieren wir das Erzwingen der Ganzzahligkeit des Arguments von T und analysieren die Rekursionsgleichung

  $$T(n) = 2T(n/2) + n$$

  Diese Änderung wirkt sich asymptotisch nicht aus.

- *Ignorieren der Anfangswertbedingung.*

  Die Lösungsmenge einer Rekursionsgleichung ohne Angabe einer Anfangswertbedingung ist eine Funktionenschar. Die (potenzielle) Anfangswertbedingung legt meist nur den konkreten konstanten Faktor bei der Lösungsfunktion fest. Da wir diese konstanten Faktoren bei der (z.B.) Θ-Notation ignorieren, brauchen wir auch keine Anfangswertbedingungen zu betrachten.

- *Abschätzungen gelten ohne besondere Erwähnung nur für große n.*

  Oftmals schätzen wir beispielsweise „$5 + \ln(n+7) \leq n$“ ab, ohne zu sagen, dass diese Abschätzung etwa erst ab $n = 8$ gilt. Solche Angaben können natürlich leicht nachgeholt werden. Es kommt jedoch bei asymptotischen Abschätzungen, zum Beispiel im Sinne der $\mathcal{O}$-Notation, nur daruaf an, dass ein entsprechender Anfangswert n_0 *existiert*.

- *Asymptotische Notation bereits in der Rekursionsgleichung.*

  Da uns am Ende nur eine Lösungsfunktion T der Form $T(n) = \Theta(\dots)$ interessiert, können wir die asymptotische Notation auch schon in der Rekursionsgleichung selber verwenden (Beispiel: $T(n) = \Theta(n) + T(n/2)$) und dadurch die Analyse wesentlich vereinfachen.

  Eine gewisse Vorsicht ist hierbei aber vonnöten. Betrachten wir zum Beispiel die iterative Einsetzungsmethode:

$$\begin{aligned} T(n) &= \Theta(n) + T(n/2) \\ &= \Theta(n) + \Theta(n/2) + T(n/4) \\ &\vdots \\ &= \Theta(n) + \Theta(n/2) + \Theta(n/4) + \dots \end{aligned}$$

  Bei dem letzten Ausdruck muss man nun aber wissen, dass die in den verschiedenen Θ-Notationen versteckten Konstanten *immer dieselben* sind! Man darf also den letzten Ausdruck nicht umformen zu

$$\Theta(n) + \Theta(n) + \Theta(n) + \dots$$

- *Obere oder untere Schranke.*

  Meist interessiert uns nicht die exakte Lösung der Rekursionsgleichung, sondern nur eine Abschätzung nach oben oder nach unten (meist nach oben). Daher kann es einfacher sein, bei einer gegebenen Rekursionsgleichung von $T(n)$ nachzuweisen, dass $T(n) \leq f(n)$ für eine gewisse Funktion f gilt als die exakte Lösung g mit $T(n) = g(n)$ zu bestimmen.

  (Möglicherweise liegt die Rekursions„gleichung“ auch nur in Form einer Ungleichung vor; Beispiel: $T(n) \leq 3T(n/3) + 2$).

1.6 Master-Theorem

Das folgende „*Master-Theorem*“ gibt die Lösung einer ganzen Klasse von Rekursionsgleichungen, wie sie typischerweise bei divide-and-conquer Algorithmen auftreten, auf einmal an.

SATZ (Master-Theorem):
Gegeben sei eine Rekursionsgleichung der Form

$$T(n) = \sum_{i=1}^{m} T(\alpha_i n) + \Theta(n^k)$$

wobei $0 < \alpha_i < 1$, $m \geq 1$, $k \geq 0$.
Dann kann $T(n)$ *asymptotisch wie folgt abgeschätzt werden:*

$$T(n) = \begin{cases} \Theta(n^k), & \text{falls } \sum_{i=1}^{m} \alpha_i^k < 1 \\ \Theta(n^k \log n), & \text{falls } \sum_{i=1}^{m} \alpha_i^k = 1 \\ \Theta(n^c), & \text{falls } \sum_{i=1}^{m} \alpha_i^k > 1 \end{cases}$$

Hierbei ist c *Lösung der Gleichung* $\sum_{i=1}^{m} \alpha_i^c = 1$. *(Falls der Spezialfall vorliegt, dass alle* α_i *denselben Wert* α *haben, so ist* $c = -\frac{\ln(m)}{\ln(\alpha)}$.*)*

Diese Rekursionsgleichung beschreibt die Laufzeit eines Algorithmus nach dem divide-and-conquer Prinzip, der die Eingabe der Größe n in m Teilprobleme der Größe $\alpha_1 n, \alpha_2 n, \ldots, \alpha_m n$ zerlegt; diese dann durch entsprechende rekursive Aufrufe löst, und aus den erhaltenen Teillösungen die Gesamtlösung zusammensetzt. Hierbei ist $\Theta(n^k)$ der Aufwand für diesen Algorithmus – ohne die rekursiven Aufrufe; dies ist also diejenige Zeit, die für das Zerlegen in Teilprobleme und für das Zusammensetzen der Gesamtlösung benötigt wird.

Beispiel:
Die drei Rekursionsgleichungen

$$\begin{aligned} T(n) &= 8T(n/3) + n^2 \\ T(n) &= 9T(n/3) + n^2 \\ T(n) &= 10T(n/3) + n^2 \end{aligned}$$

haben der Reihe nach die Lösungen:

$$\begin{aligned} T(n) &= \Theta(n^2) \\ T(n) &= \Theta(n^2 \log n) \\ T(n) &= \Theta(n^c) \end{aligned}$$

wobei $c = -\ln(10)/\ln(1/3) = \log_3 10 \approx 2.096$.

Kapitel 2

Sortier- und Selektionsalgorithmen

Sortieralgorithmen gehören zu den am häufigsten vorkommenden in der Informatik.

Gegeben sei eine Folge von Zahlen $a[1], \ldots, a[n]$. Ein *Sortieralgorithmus* soll diese Folge sortieren, das heißt, dieser soll durch entsprechende Vergleichsoperationen am Ende eine Permutation $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ bestimmen mit $a[\pi(1)] \leq a[\pi(2)] \leq \cdots \leq a[\pi(n)]$. (Tatsächlich wird diese Permutation dadurch implizit angegeben, dass die a-Folge durch den Algorithmus so durch Vertauschungen umarrangiert wird, dass sie aufsteigend sortiert ist).

Ein *Selektionsalgorithmus* bei einer gegebenen Zahlenfolge $a[1..n]$ und $i \in \{1, \ldots, n\}$ feststellen, welches das i-kleinste Element der Folge ist. (Spezialfälle hiervon sind das Bestimmen des größten Elements ($i = n$), des kleinsten Elements ($i = 1$) oder des Medians der Folge ($i = \lfloor \frac{n}{2} \rfloor$)). Man könnte die Selektionsaufgabe natürlich dadurch lösen, dass man die gesamte Folge zunächst sortiert, und dann feststellt, welches Element der Folge an die i-te Position gesetzt wurde. Aber wir werden noch sehen, dass es effizienter geht.

Bei Sortier- und Selektionsalgorithmen wird oft ein besonderes Komplexitätsmaß betrachtet, nämlich $V(n)$, die Anzahl der Vergleiche zwischen je zwei Elementen $a[i]$ und $a[j]$, also die *Vergleichskomplexität*. Die Vergleichskomplexität kann, wie üblich, sowohl im worst-case als auch im average-case analysiert werden, wobei für die average-case Analyse eine Gleichverteilung auf allen $n!$ möglichen Anordnungen der Elemente $a[1], \ldots, a[n]$ angenommen wird. Diese Funktion $V(n)$ lässt sich bei vielen Algorithmen exakt bestimmen oder gut von unten oder oben eingrenzen. Sofern $V(n)$ exakt bestimmt oder abgeschätzt wurde, so ist die tatsächliche Laufzeit-Komplexität $\Theta(V(n))$, da jeder Vergleich mit einem konstanten „Overhead“ an weiteren algorithmischen Operationen einhergeht.

2.1 BubbleSort

Es gibt verschiedene sehr simple, allerdings nicht optimale Sortier-Algorithmen. Stellvertretend betrachten wir *BubbleSort:*

Algorithmus 2.1 `bubbleSort`(a)

```
 1: t := n − 1
 2: REPEAT
 3:    m = t
 4:    b :=TRUE
 5:    FOR i := 1 TO m DO
 6:       IF a[i] > a[i + 1] THEN
 7:          b :=FALSE
 8:          t := i − 1
 9:          swap(a[i], a[i + 1])
10: UNTIL b
```

Der Name BubbleSort erklärt sich so, dass sich die Array-Elemente wie „Blasen“ durch das Array bewegen, bis sie ihre endgültige Position erreichen. Man beachte, dass BubbleSort ein *stabiles* Sortierverfahren ist, das heißt, dass gleichgroße Elemente in ihrer relativen Anordnung im Verlauf des Sortiervorgangs nicht vertauscht werden. Dies spielt bei manchen Anwendungen eine Rolle.

Günstigstenfalls ist das Array a bereits sortiert; in diesem Fall stellt dies der Algorithmus nach einem Durchlauf fest, also gilt dann $V(n) = n - 1$.

Im ersten FOR-Schleifendurchlauf (welcher $n - 1$ Vergleiche kostet) erreicht das größte Element seine endgültige Position, nämlich n; (spätestens) im zweiten Durchlauf erreicht das zweitgrößte die Position $n-1$, usw., so dass nach höchstens $n-1$ Durchläufen der FOR-Schleife die Folge sortiert ist. Wenn in einem Durchlauf die letzte Austauschaktion zwischen den Elementen $a[i]$ und $a[i+1]$ stattfand, dann befinden sich im Arrayabschnitt $a[i..n]$ bereits alle größten Elemente in sortierter Reihenfolge. Der nachfolgende Durchlauf braucht diese Elemente nicht mehr anzutasten. (Dies erklärt die Rolle der Variablen m). Also gilt für die worst-case Vergleichskomplexität

$$V(n) \leq \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

Diese Abschätzung ist scharf, denn im schlechtesten Fall befindet sich das kleinste Element der Folge a an der letzten Position n und wandert dann in jedem Durchlauf der FOR-Schleife nur um eine Position nach vorne, so dass alle $n - 1$ FOR-Schleifendurchläufe auszuführen sind. Also ergibt sich für die worst-case Vergleichskomplexität $V(n) = \frac{n(n-1)}{2}$.

Was den mittleren Fall betrifft: für alle $i \in \{1, \ldots, n\}$ gilt, dass das kleinste Element der a-Folge mit Wahrscheinlichkeit $\frac{1}{n}$ gerade das Element $a[i]$ ist. In diesem Fall ergeben sich also (mit Wahrscheinlichkeit $\frac{1}{n}$) mindestens i FOR-Schleifendurchläufe. Für die mittlere Vergleichskomplexität gilt also die Abschätzung

$$V(n) \geq \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{i(i-1)}{2} = \frac{(n+1)\cdot(n-1)}{6}$$

Bis auf den konstanten Faktor stimmt also der worst-case mit dem average-case überein, nämlich $\Theta(n^2)$.

2.2 Eine untere Schranke für (fast) alle Sortierverfahren

Für die folgende Betrachtung nehmen wir an, dass die einzige Art, wie der potenzielle Sortieralgorithmus etwas über die Anordnung der Eingabefolge erfährt, darin besteht, Abfragen der Form „$a[i] < a[j]$“ zu tätigen. Der Algorithmus BubbleSort (und auch die noch zu besprechenden Algorithmen QuickSort, MergeSort, HeapSort) ist von dieser Art. Da wir keine Annahme über die absolute Größe der zu sortierenden Zahlen treffen, ist eine (hier nicht zulässige) Abfrage wie zum Beispiel „$a[3] > 100$“ auch nicht sinnvoll.

Jeder Sortieralgorithmus ist zunächst in vollständiger „Unwissenheit“ darüber, welche Permutation π die gegebene Eingabe $a[1..n]$ sortiert (das heißt, für welches π gilt, dass $\pi(i)$ gerade der Rang des Elements $a[i]$ ist, $i = 1, 2, \ldots, n$). Die Menge der zu Beginn der Rechnung möglichen Permutationen U_0 hat die Mächtigkeit $n!$. Durch eine IF-Abfrage nach „$a[i] < a[j]$“ wird die Menge der möglichen Permutationen in zwei Teilmengen zerlegt. Genauer: Sei U_k die Menge der möglichen Permutationen nach dem k-ten Vergleich. Nach Ausführung des Vergleichs „$a[i] < a[j]$“ wird U_k zerlegt in

$$\begin{aligned} U'_k &= \{\pi \in U_k \mid \pi(i) < \pi(j)\} \\ U''_k &= \{\pi \in U_k \mid \pi(i) > \pi(j)\} \end{aligned}$$

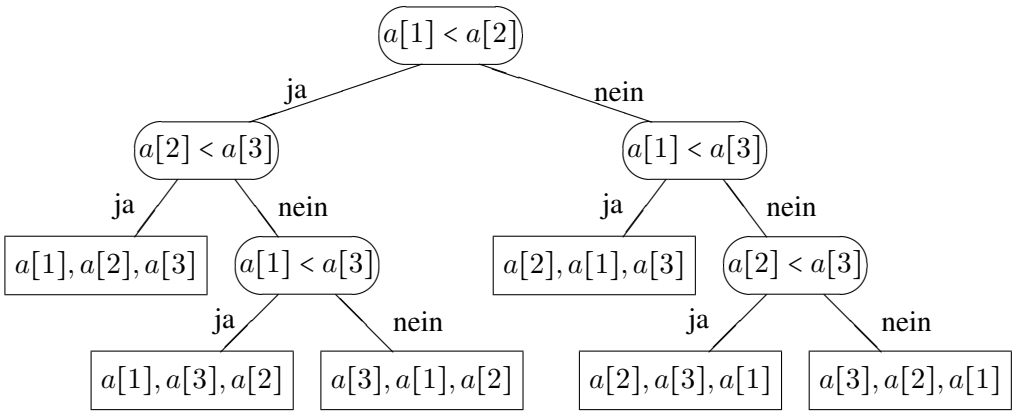
Je nachdem, wie der Vergleich ausfällt, gilt dann $U_{k+1} = U'_k$ oder $U_{k+1} = U''_k$. Im schlechtesten Fall liegt die gesuchte Permutation π in der größeren der beiden Mengen U'_k oder U''_k. Daher kann sich im schlechtesten Fall $|U_{k+1}|$ gegenüber $|U_k|$ höchstens halbiert haben, also $|U_{k+1}| \geq \frac{|U_k|}{2}$. Jeder Sortieralgorithmus muss schlimmstenfalls so viele Vergleiche machen, wie man benötigt, durch fortgesetztes Halbieren die Zahl $|U_0| = n!$ zu einer Zahl ≤ 1 zu machen. Dies sind also mindestens $\lceil \log_2(n!) \rceil$ Vergleiche. (Denn wenn ein Sortieralgorithmus nach k Schritten stoppt, solange sich noch mehr als eine Permutation in U_k befindet, ist dieser inkorrekt, denn er behandelt dann mindestens zwei Eingabepermutationen identisch).

Asymptotisch verhält sich $\log_2(n!)$ wie $n \log_2 n - \Theta(n)$ (vgl. Abschnitt 1.1). Daher muss *jeder* (auf paarweisen Element-Vergleichen basierende) Sortieralgorithmus die worstcase Vergleichskomplexität $\geq n \log n - \Theta(n)$ haben.

Hinter dem gerade gegebenen Beweis steckt das Konzept des *Entscheidungsbaums*. Jede Verzweigung (=innerer Knoten des Baumes) ist mit einer Abfrage „$a[i] < a[j]$“ beschriftet. An den Blättern des Entscheidungsbaums ist die festgestellte Permutation eingetragen. Da es $n!$ verschiedene Permutationen gibt, muss der einem Sortieralgorithmus zugeordnete Entscheidungsbaum mindestens $n!$ Blätter haben. Dies impliziert, dass es mindestens einen Pfad der Länge $\lceil \log_2(n!) \rceil$ in dem Baum gibt. (Diesen Zusammenhang zwischen der Anzahl der Blätter und der Tiefe eines Binärbaumes sieht man sofort mit einer Induktion ein).

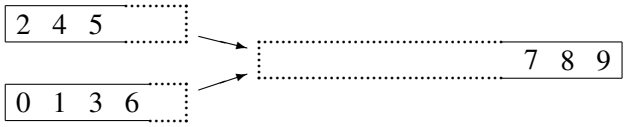
Man spricht in diesem Zusammenhang auch von der *informationstheoretischen Schranke*, denn um durch binäre ja/nein-Fragen zwischen einer von $n!$ Möglichkeiten zu unterscheiden, benötigt man (schlechtestenfalls) $\lceil \log_2(n!) \rceil$ viele Fragen.

Beispiel:
Der folgende spezielle Entscheidungsbaum „sortiert“ das Array $a[1..3]$. Da $3! = 6$, muss dieser mindestens 6 Blätter besitzen. Da $\lceil \log_2 6 \rceil = 3$, muss dieser Baum einen Pfad der Länge 3 besitzen.



2.3 MergeSort

In der rekursiven Formulierung ist MergeSort ein klassisches divide-and-conquer Verfahren, das genau dem Schema auf Seite 12 entspricht. Das zu sortierende Array $a[1..n]$ wird zunächst in zwei gleich große Hälften aufgeteilt; diese werden separat durch zwei rekursive Aufrufe sortiert. Die beiden sortierten Teilarrays werden sodann zu einer sortierten Gesamtfolge „gemischt“. Den Mischvorgang kann man sich (wie bei einem Reißverschluss) wie folgt veranschaulichen. Wir zeigen eine Momentaufnahme. Die beiden sortierten Teilfolgen $(2, 4, 5, 7, 8)$ und $(0, 1, 3, 6, 9)$ der Länge $n/2$ wurden bereits zum Teil in ein weiteres Array in sortierter Reihenfolge hineingeschrieben.



Im nächsten Schritt müßte die Zahl 5 mit der Zahl 6 verglichen werden, und die größere der beiden, die 6, wandert dann in das Ergebnis-Array, usw.

Hier ist nochmals der Algorithmus:

Algorithmus 2.2 `mergeSort`(a)

```
1: IF |a| = 1 THEN
2:    return
3: ELSE
4:    Zerlege a in zwei gleichgroße Teile a' und a''
5:    mergeSort(a')
6:    mergeSort(a'')
7:    Mische a' und a'' zu einem Array b
8:    a := b
9:    return
```

Dieser Mischvorgang benötigt $n-1$ Vergleiche zwischen Elementen. Daher erhalten wir die Rekursionsgleichung $V(n) = 2V(n/2) + n - 1$, $V(1) = 0$.

$$\begin{aligned}
V(n) &= 2V\left(\frac{n}{2}\right) + n - 1 \\
&= 2 \cdot (2V\left(\frac{n}{4}\right) + \frac{n}{2} - 1) + n - 1 \\
&= 4V\left(\frac{n}{4}\right) + n - 2 + n - 1 \\
&= 8V\left(\frac{n}{8}\right) + n - 4 + n - 2 + n - 1 \\
&= \log_2 n \cdot n - \sum_{i=0}^{\log_2 n} 2^i \\
&= n \log_2 n + \frac{1 - 2^{\log_2(n)+1}}{1} \\
&= n \log_2 n + 1 - 2 \cdot 2^{\log_2 n} \\
&= n \log_2 n + 1 - 2n
\end{aligned}$$

Verglichen mit der unteren Schranke $V(n) \geq \log_2(n!) = n \log_2 n - \frac{n}{\ln 2} + \Theta(\log n)$ ist dies, was die Anzahl der Vergleiche betrifft, optimal (abgesehen von einem Term linearer Ordnung).

Ein Nachteil von MergeSort ist, dass in ein separates Array umgespeichert werden muss. Es ist also kein *in-place* Verfahren, welches nur das eigentliche Array und höchstens konstant viel zusätzlichen Speicherplatz für die Variablen benötigt.

Andererseits ist MergeSort (in nicht-rekursiver Implementierung) ein sehr geeignetes *externes* Sortierverfahren. Das heißt, wenn man davon ausgehen muss, dass die zu sortierenden Daten nicht als Ganzes im Hauptspeicher untergebracht werden können, sondern in einer externen Datei untergebracht sind, so eignet sich MergeSort zur Sortierung, indem man insgesamt 3 Dateien verwendet.

Wenn man sich die Rekursion bei MergeSort „abgespult“ vorstellt, und entsprechend des Rekursionsstruktur von unten nach oben vorgeht, so bedeutet dies, dass MergeSort aus der Eingabefolge, welche als n sortierte Teilfolgen der Länge 1 aufgefasst wird, $n/2$ viele sortierte Teilfolgen der Länge 2 herstellt, dann $n/4$ viele sortierte Folgen der Länge 4, usw. bis man nach $\log_2 n$ Durchgängen durch die Dateien fertig ist. Eine bessere Implementierung, die nicht stur an den Zweierpotenzen festhält und bereits sortierte Teilfolgen in der Eingabe berücksichtigt, ist das sog. *natürliche Mischen*.

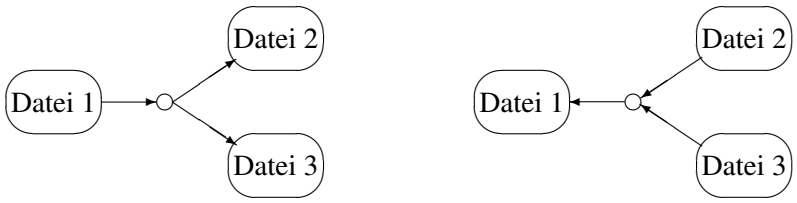
Hierbei werden die Zahlen, die zunächst in Datei 1 stehen, so alternierend in die Dateien 2 und 3 umgeschrieben, dass bereits aufsteigend sortierte Teilfolgen (sog. *Runs*) zusammen bleiben und hintereinander in Datei 2 (bzw. 3) geschrieben werden. Erst wenn der Run unterbrochen ist, weil eine zu kleine Zahl nachfolgt, wird von Datei 2 auf Datei 3 gewechselt (oder umgekehrt).

Algorithmus 2.3 `divide(Datei1)`

```
1: k := 2
2: Reset(Datei1); Rewrite(Datei2); Rewrite(Datei3);
3: WHILE !eof(Datei1) DO
4:    Schreibe den nächsten Run von Datei1 nach Datei k
5:    IF k == 2 THEN
6:       k := 3
7:    ELSE
8:       k := 2
```

Nach diesem Zerlege-Vorgang folgt der Misch-Vorgang, der von den Dateien 2 und 3 wieder zurück in die Datei 1 führt. Dabei werden je zwei aufsteigend sortierte Runs, einer aus Datei 2 und einer aus Datei 3, zu einer aufsteigend sortierten Teilfolge zusammengemischt und dabei in Datei 1 geschrieben. Das Ende eines Runs in Datei 2 bzw. 3 kann wieder daran erkannt werden, dass eine kleinere Zahl als die zuvor gelesene nachfolgt.

Skizze:



Schlimmstenfalls ist die Ausgangsfolge umgekehrt sortiert, dann entstehen genau wie oben beschrieben, sortierte Teilfolgen der Länge 1, 2, 4, 8, usw. Günstigenfalls ist die Folge bereits sortiert. Dies wird beim Zerlege-Vorgang festgestellt und das Verfahren kann dann sofort stoppen. Insgesamt ergibt sich auf jeden Fall ein $\Theta(n \log n)$ Verfahren.

2.4 QuickSort

QuickSort ist ein im Mittel sehr schnelles Sortierverfahren; es erreicht im average-case die bestmögliche Komplexität, die, wie wir gesehen haben, $\Theta(n \log n)$ beträgt.

Im worst-case hat QuickSort allerdings wie die naiven Verfahren die Komplexität $\Theta(n^2)$.

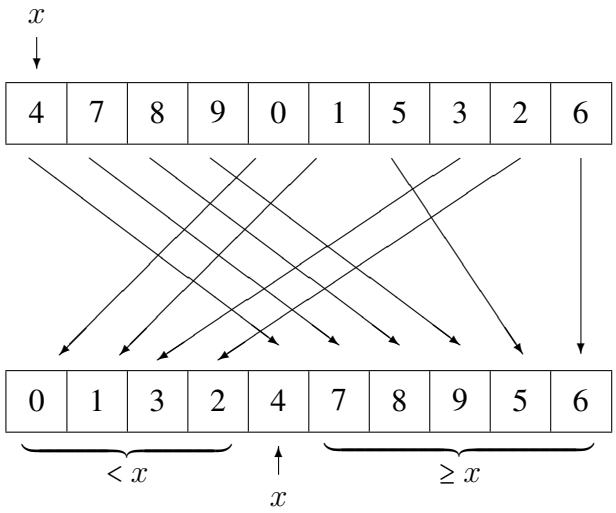
QuickSort, ein typisches *divide-and-conquer* Verfahren, funktioniert wie folgt. Im Unterschied zu MergeSort entsteht der eigentliche Aufwand *vor* Ausführung der rekursiven Aufrufe; bei MergeSort entsteht dieser *danach*.

Algorithmus 2.4 `quickSort(links, rechts)`

```
1: //Sortiert den Array-Abschnitt a[links..rechts]
2: IF rechts−links ≥ 1 THEN
3:    Wähle ein Element x aus a[links..rechts], z.B. x = a[links]
4:    Ordne a[links..rechts] so um, dass alle Array-Elemente, die kleiner (größer) als x
      sind, links (rechts) von x angeordnet werden.
5:    Sei q die Array-Position, die x bei der Umordnung von a erhält.
6:    quickSort(links, q − 1)
7:    quickSort(q + 1, rechts)
```

Im Hauptprogramm wird diese Prozedur durch `quickSort(1, n)` aufgerufen.

Die folgende Skizze zeigt ein Beispiel für den Aufteilungsprozess des Arrays.



Bei diesem Beispiel ist das erste Array-Element das Aufteilungselement x (das „Pivotelement“), nämlich die 4. Die entstehende Zerlegung ist dann solcherart, dass im linken Teil-Array noch 4 Elemente zu sortieren sind, im rechten Teil-Array noch 5.

Im *schlechtesten Fall* gilt bei jedem rekursiven Aufruf q = links (oder q = rechts). In diesem Fall sind im ersten Aufruf von QuickSort $n-1$ Elemente-Vergleiche notwendig, dann im nächsten $n-2$, usw. Wenn $V(n)$ die Anzahl der Vergleichsoperationen im schlechtesten Fall ist, so gilt also (wie bei BubbleSort)

$$V(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

Also ist wc-time$_{\texttt{quicksort}}(n) = \Theta(n^2)$. (Interessanterweise tritt dieser schlechteste Fall unter anderem dann auf, wenn die zu sortierende Zahlenfolge bereits sortiert ist!)

Im *günstigsten Fall* werden bei jedem Umordnungs-Vorgang zwei gleichgroße Hälften geschaffen. Dies ist dann der Fall, wenn das Element x, das sog. *Pivotelement* gerade der Median des Arrays a[links..rechts] ist.

Sei $V(n)$ die Anzahl der Vergleiche zum Sortieren von n Elementen. Wenn also jedes Mal dieser günstigste Fall vorliegt, dann gilt die Rekursionsgleichung

$$V(n) = n - 1 + 2 \cdot V\left(\frac{n}{2}\right), \quad V(1) = 0$$

denn $n - 1$ Vergleiche werden für das Aufteilen selbst benötigt. Mit den Methoden aus Abschnitt 1.4 ergibt sich $V(n) = n \cdot \log_2 n + \Theta(n)$. Im günstigsten Fall erreicht man also (bis auf den $\Theta(n)$-Term) die minimal mögliche Anzahl von Vergleichen.

Wir wollen nun das *average-case Verhalten* von QuickSort analysieren. Wir nehmen hierzu an, dass alle $n!$ Permutationen der Eingabezahlen gleichwahrscheinlich sind.

Sei s_i dasjenige Array-Element, das den Rang i hat, das heißt, s_i ist das i-kleinste Element in a. Sei X_{ij} $(i < j)$ eine 0-1-wertige Zufallsvariable, die genau dann den Wert 1 annimmt, wenn die Elemente s_i und s_j bei einem Lauf von QuickSort – bei zufälliger Eingabefolge $a[1..n]$ – miteinander verglichen werden. (Man überzeuge sich davon, dass zwei Array-Elemente bei QuickSort höchstens einmal miteinander verglichen werden). Dann ist die mittlere Anzahl von Vergleichen, die QuickSort bei zufälliger Eingabe tätigt, gegeben durch

$$V(n) = \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}[X_{ij}]$$

Sei p_{ij} die Wahrscheinlichkeit, dass bei einem Lauf von QuickSort die Elemente s_i und s_j verglichen werden. Da

$$\mathbb{E}[X_{ij}] = 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) = p_{ij}$$

besteht die Aufgabe, $V(n)$ zu berechnen, letztlich darin, p_{ij} zu berechnen.

Behauptung:
Genau die folgenden Eingabe-Permutationen führen dazu, dass ein Vergleich zwischen s_i und s_j $(i < j)$ stattfindet: Im Eingabe-Array $a[1..n]$ sei von den Zahlen $s_i, s_{i+1}, \ldots, s_j$ die Zahl s_μ diejenige mit dem kleinsten Index, also $s_\mu = a[k]$, k minimal. Es findet genau dann ein Vergleich zwischen s_i und s_j statt, wenn $\mu = i$ oder $\mu = j$.

Daher gilt für die gesuchte Wahrscheinlichkeit $p_{ij} = \frac{\#\text{ günstige Fälle}}{\#\text{ alle Fälle}} = \frac{2}{|\{s_i,\ldots,s_j\}|} = \frac{2}{j-i+1}$, denn in genau 2 von $j-i+1$ gleichwahrscheinlichen Fällen tritt die geschilderte Situation ein.

Die Begründung für die obige Behauptung ist wie folgt: In jedem Aufteilungsvorgang des Arrays, in dem das Pivotelement nicht aus dem Abschnitt $s_i, \ldots, s_j$ stammt, wird die relative Reihenfolge der Elemente $s_i, \ldots, s_j$ im betreffenden Teilarray nicht geändert. (Zur Veranschaulichung vgl. das Bild auf Seite 25). Dasjenige Element mit dem kleinsten Index bleibt also das mit kleinstem Index. Insbesondere werden in diesem Fall s_i und s_j nicht miteinander verglichen. Wenn schließlich in einem rekursiven Aufruf von QuickSort ein Pivotelement x aus der Menge $\{s_i, \ldots, s_j\}$ ausgewählt wird, so ist es dasjenige s_μ, das im Eingabe-Array den kleinsten Index hat.

Sofern das Pivotelement x aus der Menge $\{s_i, \ldots, s_j\}$ stammt, und $x \neq s_i$ und $x \neq s_j$, so geraten s_i und s_j in verschiedene Teilarrays und werden im späteren Verlauf nicht mehr

miteinander verglichen. Nur im Fall, dass $x = s_i$ oder $x = s_j$ werden die beiden Elemente im Laufe des Partitionsvorgangs miteinander verglichen.

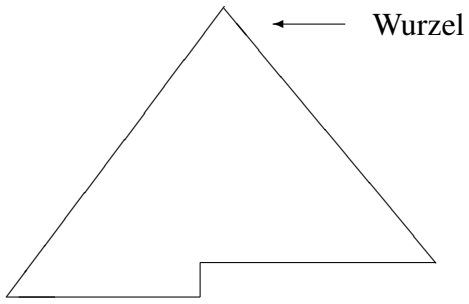
Wir setzen nun in die oben entwickelte Formel ein und erhalten für die gesuchte mittlere Anzahl von Vergleichen:

$$\begin{aligned} V(n) &= \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} p_{ij} = \sum_{i<j} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq \sum_{i=1}^{n-1} 2(H_n - 1) \leq 2n(H_n - 1) \\ &\leq 2n \ln n = (2 \ln 2) n \log_2 n \leq 1.3863 \cdot n \log_2 n \end{aligned}$$

Hierbei ist H_n die harmonische Reihe. Im Mittel benötigt QuickSort also etwa 39 Prozent mehr Vergleiche als das potenzielle Optimum. Auf jeden Fall ist die average-case Komplexität von QuickSort von der Größenordnung $\Theta(n \log n)$.

2.5 HeapSort

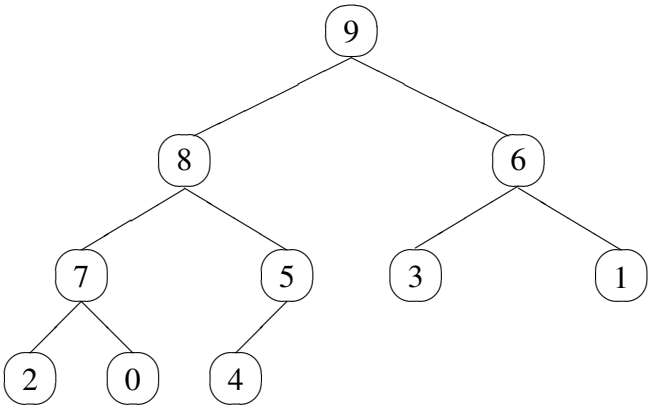
HeapSort operiert auf einer Datenstruktur, die *Heap* genannt wird. Da dieser Heap absolut im Mittelpunkt des Algorithmus steht und die gesamte Vorgehensweise durch diese Datenstruktur bestimmt wird, ist HeapSort ein Beispiel für einen „Datenstrukturgetriebenen“ Algorithmus. Ein Heap ist ein Binärbaum, der – soweit es geht – vollständig aufgefüllt ist. Nur die unterste Schicht ist möglicherweise nicht voll aufgefüllt. Was diese Schicht betrifft, so muss diese *linksbündig* aufgefüllt werden. Die Struktur eines Heaps sieht also wie folgt aus:



Die Anzahl der Knoten in einem Heap legt also eindeutig dessen Struktur fest.

Jeder Knoten des Heaps ist ferner mit einer Zahl beschriftet. Diese Beschriftung muss solcherart sein, dass der Vaterknoten immer einen höheren Zahlenwert trägt als jeder seiner Söhne. Zwischen den beiden Söhnen braucht keine Relation eingehalten zu werden.

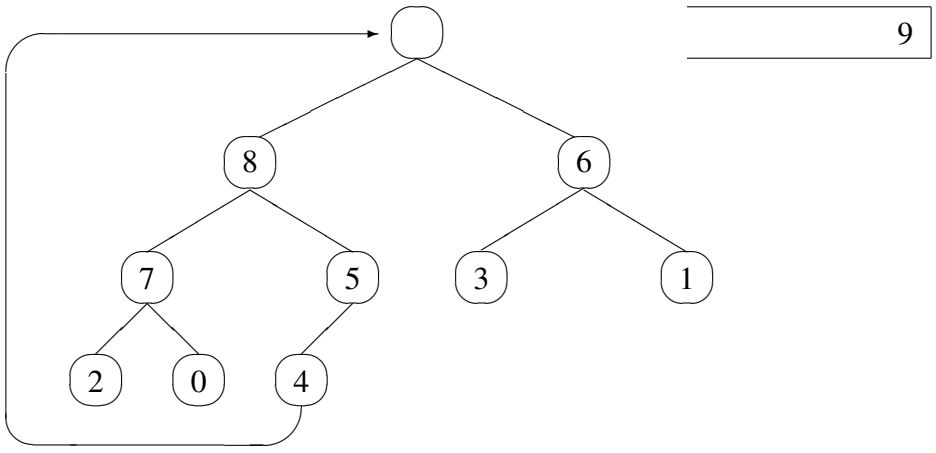
Beispiel:
Das Folgende ist ein Heap mit 10 Knoten, der mit den Zahlen 0,1,..., 9 in einer möglichen Weise beschriftet ist:



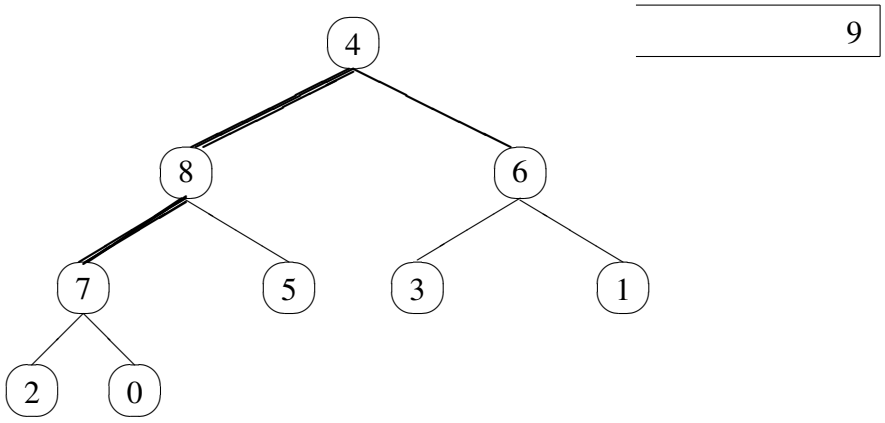
Der Wert der Wurzel muss notwendigerweise der größte der Zahlenmenge sein.

HeapSort arbeitet nun in zwei Phasen: in der ersten Phase wird aus der gegebenen Zahlenfolge ein Heap aufgebaut. In der zweiten Phase wird der Heap wieder abgebaut, dabei entsteht sukzessive eine sortierte Zahlenfolge.

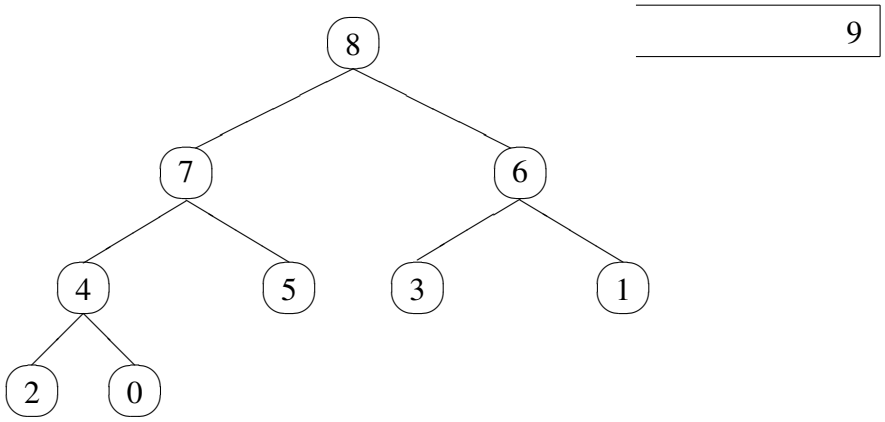
Betrachten wir zunächst die zweite Phase. Bei dem obigen Beispiel-Heap steht die größte Zahl gerade an der Wurzel. „Pflücken“ wir die Zahl an der Wurzel ab, so ist natürlich die Heap-Eigenschaft verletzt:



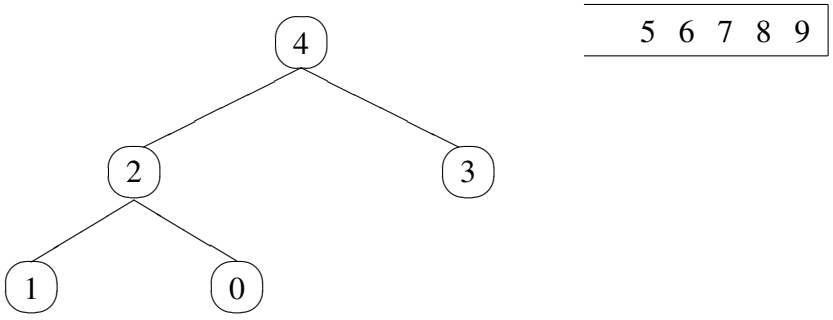
Wir setzen nun das Element der untersten Schicht, das am weitesten rechts liegt, an die Wurzelposition. Nun ist zwar die Heap-Baumstruktur wieder hergestellt, aber die Heap-Bedingung (Vater $\geq$ Sohn) ist an der Wurzel noch nicht erfüllt. Wir folgen nun, beginnend bei der Wurzel, dem Pfad entlang des jeweils größeren der beiden Söhne, bis wir entweder auf Blattebene ankommen oder bei einem Knoten, dessen Söhne mit kleineren Werten als der momentane Wurzelwert beschriftet sind.



Dadurch haben wir eine Folge von Knoten gefunden, deren Werte nur im Ringtausch ausgewechselt werden müssen, um die Heap-Eigenschaft wieder herzustellen.



So geht es weiter. Nach 4 weiteren solchen Schritten erhalten wir:

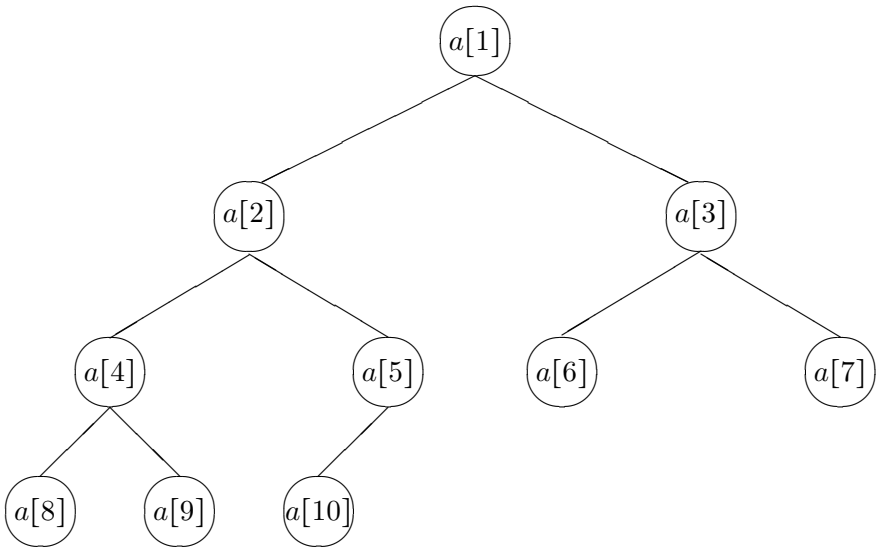


Auf diese Weise entsteht eine sortierte Folge.

Diesen Prozess des Wiederherstellens der Heap-Bedingung, wenn diese höchstens beim Wurzelknoten verletzt ist, indem man den Wurzelwert mit dem jeweils größeren der beiden Söhne vertauscht, nennen wir *Heapify*.

Bis zu diesem Zeitpunkt sieht es so aus als benötigten wir zur Realisierung dieses Algorithmus eine verzeigerte Datenstruktur, um den Heap zu realisieren. Der Heap kann jedoch in einem Array repräsentiert werden, und zwar spiegelt sich die Heap-Struktur

im Array durch die Indizierung wider:



Das heißt, die Vater-Sohn-Beziehung lässt sich aus der Array-Indizierung wie folgt entnehmen: Der Knoten $a[i]$ hat die Söhne $a[2i]$ und $a[2i+1]$ (sofern $2i$ bzw. $2i+1$ kleinergleich n ist). Umgekehrt hat der Knoten $a[k]$ (für $2 \leq k \leq n$) den Vater $a[k \text{ div } 2]$.
Die Heap-Bedingung lautet dann also

$$a[i] \geq a[2i] \quad \text{und} \quad a[i] \geq a[2i+1]$$

Wir wollen im Folgenden die Heap-Bedingung (per Definition) auch auf einen *Teilabschnitt* des Arrays anwenden. Daher definieren wir, dass ein Abschnitt $a[li..re] \subseteq a[1..n]$ die Heapbedingung erfüllt, falls gilt:

$$\forall i \in \{li, \ldots, re\} : \Big((2i \leq re \Rightarrow a[i] \geq a[2i]) \wedge (2i+1 \leq re \Rightarrow a[i] \geq a[2i+1]) \Big)$$

Man beachte, dass der Abschnitt $a[n \text{ div } 2{+}1, \ldots, n]$ immer die Heap-Bedingung erfüllt, da es sich bei den in diesem Abschnitt vorkommenden Knoten nur um Blätter handelt.
Die Prozedur *Heapify* können wir auf der Array-Repräsentation wie folgt realisieren:

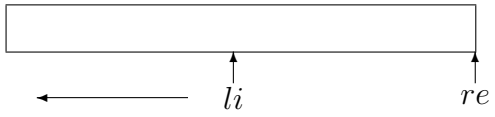
Algorithmus 2.5 `heapify`(li, re)

```
 1: k := 2 · li
 2: IF k > re THEN
 3:    return
 4: IF k + 1 > re THEN
 5:    IF a[k] > a[li] THEN
 6:       vertausche(a[li], a[k])
 7:       return
 8: IF a[i] < a[k + 1] THEN
 9:    k := k + 1
10: IF a[li] < a[k] THEN
11:    vertausche(a[li], a[k])
12:    heapify(k, re)
```

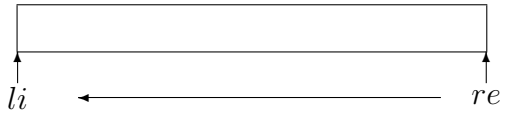
HeapSort wird nun so implementiert, dass in der ersten Phase der Heap aufgebaut wird, indem der Links-Zeiger sukzessive nach vorne verschoben wird und jedes Mal wieder mittels *Heapify* die HEAP-Bedingung hergestellt wird. In der zweiten Phase werden die Wurzelelemente „abgepflückt“, indem der Rechts-Zeiger nach links verschoben wird und das Wurzelelement jenseits des rechten Rands des Heaps umgetauscht wird.

Skizze:

Phase 1:



Phase 2:



Wir formulieren dies als Programm:

Algorithmus 2.6 `heapsort`(a, n)

```
 1: li := n DIV 2 + 1
 2: re := n
 3: //Phase 1:
 4: WHILE li > 1 DO
 5:    li := li – 1
 6:    heapify(li, re)
 7: //Phase 2:
 8: WHILE re > 1 DO
 9:    vertausche(a[re], a[li])
10:    re := re – 1
11: heapify(lie, re)
```

Zur Komplexitätsanalyse bemerken wir zunächst, dass HeapSort ohne zusätzlichen Speicheraufwand realisiert werden kann (also $\mathcal{O}(1)$ zusätzlicher Speicheraufwand). Durch die Rekursion bei *heapify* entsteht eigentlich ein gewisser Speicheraufwand; aber die Prozedur *heapify* kann ohne weiteres nicht-rekursiv und ohne zusätzlichen Speicheraufwand programmiert werden, da es sich um eine *tail recursion* handelt. Man spricht von einem *in-place* (oder „in-situ“) Sortierverfahren.

Was die worst-case Komplexität von HeapSort betrifft, so ist ziemlich offensichtlich, dass dies ein $\mathcal{O}(n \log n)$-Verfahren ist: Ein Aufruf von *heapify* erzeugt höchstens $\log_2 n$ viele Austauschaktionen (da dies die maximale Baumtiefe eines Heaps mit n Knoten ist), hat also die Komplexität $\mathcal{O}(\log n)$. Die Prozedur *heapify* muss in der ersten Phase etwa $(n/2)$-mal und in der zweiten Phase $(n - 1)$-mal aufgerufen werden, daher ist die Komplexität also insgesamt durch $\mathcal{O}(n \log n)$ beschränkt. Also ist die Komplexität von HeapSort (bis auf den konstanten Faktor in der $\mathcal{O}$-Notation) optimal – im Sinne von Abschnitt 2.2.

Wir wollen aber noch die genaue Anzahl $V(n)$ von Vergleichen abschätzen, die HeapSort benötigt, und diese mit der informationstheoretischen unteren Schranke (vgl. Abschnitt 2.2) $V(n) \geq \log_2(n!) = n \log_2 n - \Theta(n)$ vergleichen.
Pro Austauschvorgang in der Prozedur *heapify* werden 2 Vergleiche gemacht (nämlich, um das größte Element der Menge $\{a[i], a[2i], a[2i + 1]\}$ zu bestimmen). Wir werden gleich sehen, dass die Phase 1 insgesamt nur $\mathcal{O}(n)$ Vergleiche benötigt. Daher ist die Anzahl notwendiger Vergleiche von HeapSort (im schlechtesten Fall) $2n \log_2 n + \mathcal{O}(n)$, also um den Faktor 2 schlechter als das potenzielle Optimum.

Wir wollen noch zeigen, dass die Phase 1 tatsächlich nur $\mathcal{O}(n)$ Vergleiche benötigt (das heißt, die einzelnen Einfügeoperationen in der 1. Phase haben nur eine amortisierte Komplexität von $\mathcal{O}(1)$). Sei a ein Array mit n Elementen (=Knoten). Die Zahl der Vergleiche, um a in die Heapstruktur zu bringen, ist höchstens 2 mal so groß wie die Summe der Entfernungen aller Knoten bis zur Blattebene, denn dies ist die maximale Wegstrecke entlang der beim Heapaufbau Austausch- und Vergleichsoperationen stattfinden kön-

nen. Der Abstand eines Knotens von der Blattebene ist, anders ausgedrückt, die Tiefe des Teilbaums, dessen Wurzel der betreffende Knoten ist.

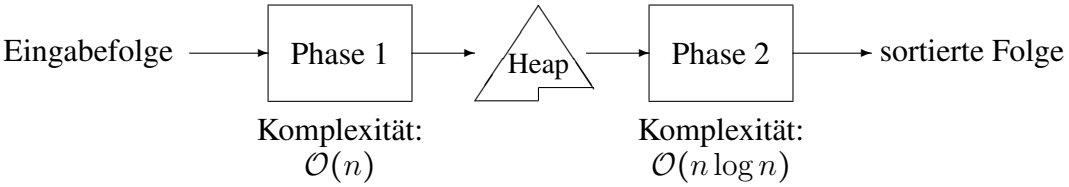
Man kann zeigen, dass bei einem Heap mit n Knoten die Summe dieser Abstände zur Blattebene die Zahl n nicht übersteigt. Und zwar sind etwa $n/2$ der Knoten Blätter, haben damit Abstand 0 zur Blattebene, etwa ein Viertel aller Knoten hat den Abstand 1, etwa ein Achtel aller Knoten hat den Abstand 2, usw.

Hierzu zwei Beispiele: bei einem vollständigem Binärbaum der Tiefe 4, welcher $n = 31$ Knoten besitzt, haben 16 Knoten den Abstand 0, 8 Knoten den Abstand 1, 4 Knoten den Abstand 2, 2 Knoten den Abstand 3 und ein Knoten (die Wurzel) den Abstand 4 von der Blattebene. In der Summe ergibt dies 26 ($< n$). Allgemein ergibt sich, dass ein vollständiger Binärbaum der Tiefe k (der $n = 2^{k+1} - 1$ Knoten besitzt), die Abstands-Summe $n - k - 1$ hat.

Wenn wir nun einen weiteren Knoten zum Heap hinzufügen (also $n = 32$), so erhöht sich für alle Knoten entlang des Pfades von der Wurzel zu diesem hinzugefügten Knoten der Abstand zur Blattebene um 1. Die neue Bilanz sieht so aus: 16 Knoten haben den Abstand 0, 8 Knoten den Abstand 1, 4 Knoten den Abstand 2, 2 Knoten den Abstand 3, ein Knoten den Abstand 4 und ein Knoten (die Wurzel) den Abstand 5 von der Blattebene. In der Summe ergibt dies 31 ($< n$). Allgemein hat ein Heap mit $n = 2^{k+1}$ Knoten die Abstands-Summe $n - 1$.

Alle anderen Heaps liegen zwischen diesen beiden Extremen.

Das folgende Bild skizziert nochmals die Phasen bei HeapSort:



2.6 BucketSort

BucketSort (auch: Sortieren durch Fachverteilen) unterbietet im average-case die $n \log n$ Schranke, denn das Verfahren hat in diesem Fall die Komplexität $\mathcal{O}(n)$. Wie kann das im Hinblick auf Abschnitt 2.2 angehen? Der Grund liegt darin, dass BucketSort sich nicht an die in Abschnitt 2.2 vorgegebenen „Spielregeln“ hält, welche nämlich fordern, dass die einzig zulässigen Vergleiche solche zwischen Elementen $a[i]$ und $a[j]$ sind.

Diese average-case Komplexität $\mathcal{O}(n)$ beruht allerdings auf einer weitergehenden Annahme über die Verteilung der Eingabezahlen: Die Werte $a[i] \in \mathbb{R}$ müssen unabhängig und gleichverteilt in einem festen Intervall $[a, b)$ mit $a < b$ sein. (Im Folgenden beschränken wir uns der Einfachheit halber auf $a = 0$ und $b = 1$).

Der Nachteil von BucketSort ist allerdings, dass es $\mathcal{O}(n)$ zusätzlichen Speicherplatz benötigt, also kein „in-place“ Verfahren ist.

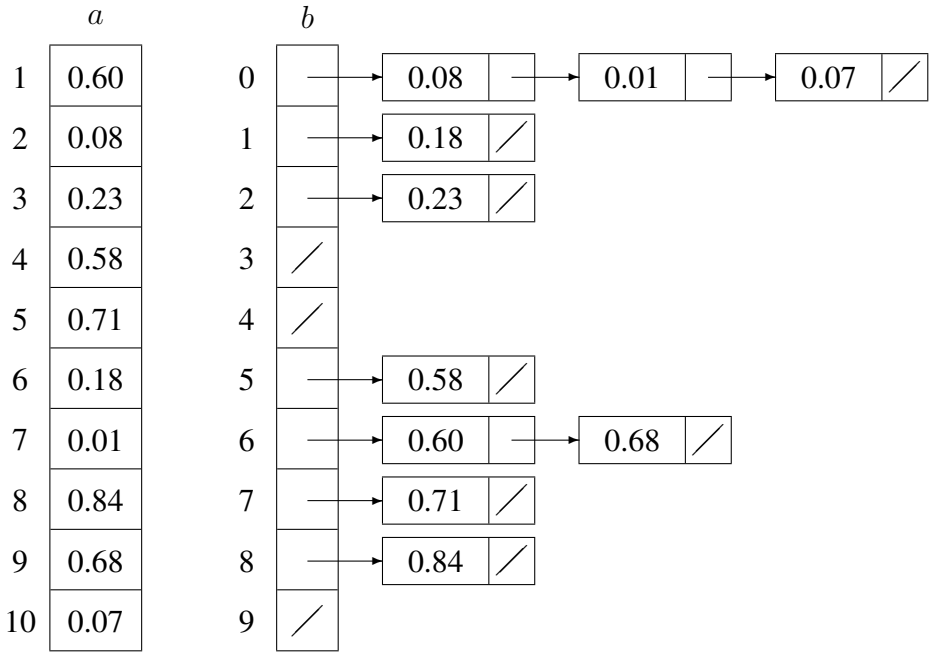
Wir beschreiben nun das Verfahren. Die Eingabe sei $a[1..n]$. BucketSort benötigt ein weiteres Array $b[0\,..\,n-1]$ von Zeigern, welches der Ausgangspunkt für n lineare, ver-

kettete Listen ist. Diese sind die „Buckets“, welche gerade solche $a[i]$-Elemente zusammenfassen, die in dasselbe Zahlenintervall fallen. Hierbei wird das Intervall $[a, b) = [0, 1)$ in n gleichgroße Intervalle zerlegt und der Bucket $b[i]$ ist „zuständig“ für das Zahlenintervall $[i/n,\ (i+1)/n)$.

Algorithmus 2.7 `bucketsort`$(a[1..n])$

1: **FOR** $i := 0$ **TO** $n - 1$ **DO**
2: Initialisiere $b[i]$ mit der leeren Liste
3: **FOR** $i := 1$ **TO** n **DO**
4: Trage $a[i]$ in die Liste $b[\lfloor n \cdot a[i] \rfloor]$ ein
5: **FOR** $i := 0$ **TO** $n - 1$ **DO**
6: Sortiere die Liste $b[i]$ nach einem Standard-Verfahren, das auf verkettete Listen anwendbar ist, z.B. BubbleSort
7: Füge die Listen $b[0], b[1], \ldots, b[n-1]$ zu einer sortierten Liste zusammen (und übertrage diese in a)

Beispiel:
Die Situation nach dem Einfügen in die Buckets, aber vor deren Sortierung, könnte wie folgt aussehen:



Es ist klar, dass im worst-case alle Elemente in ein und denselben Bucket geraten. Dann ergibt sich die Komplexität $\mathcal{O}(n^2)$.

Wir analysieren nun die average-case Komplexität: Sei X_i eine Zufallsvariable, die angibt, wie viele Elemente in „Bucket“ i fallen (X_i ist die Länge der Liste $b[i]$). Für jedes

Array-Element $a[j]$, $j = 1, \ldots, n$, gilt:

$$Pr(\, a[j] \text{ kommt in Liste } b[i]\,) \;=\; 1/n$$

Dies gilt voneinander unabhängig für alle j. Daher liegt hier ein Bernoulli-Experiment vor. Somit ist die Zufallsvariable X_i *binomialverteilt* mit den Parametern n und $p = 1/n$. Also gilt:

$$\begin{aligned} \mathbb{E}[X_i] &= np = 1 \\ V[X_i] &= np(1-p) = 1 - 1/n \end{aligned}$$

Ferner beobachten wir, dass $V[X_i] = \mathbb{E}[X_i^2] - (\mathbb{E}[X_i])^2$, also $\mathbb{E}[X_i^2] = V[X_i] + (\mathbb{E}[X_i])^2 = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n}$.
Der Aufwand für das Sortieren von Liste $b[i]$ ist $\mathcal{O}(X_i^2)$. Insgesamt ergibt dies somit

$$\sum_{i=0}^{n-1} \mathcal{O}(X_i^2)$$

Der mittlere Gesamtaufwand zum Sortieren aller $b[i]$'s ist somit

$$\begin{aligned} \mathbb{E}\Big[\sum_{i=0}^{n-1} \mathcal{O}(X_i^2)\Big] &= \mathbb{E}\Big[\mathcal{O}\Big(\sum_{i=0}^{n-1} X_i^2\Big)\Big] = \mathcal{O}\Big(\mathbb{E}\Big[\sum_{i=0}^{n-1} X_i^2\Big]\Big) \\ &= \mathcal{O}\Big(\sum_{i=0}^{n-1} \mathbb{E}[X_i^2]\Big) = \mathcal{O}\Big(\sum_{i=0}^{n-1} (V[X_i] + (\mathbb{E}[X_i])^2)\Big) \\ &= \mathcal{O}\Big(\sum_{i=0}^{n-1} (2 - 1/n)\Big) = \mathcal{O}(2n-1) = \mathcal{O}(n) \end{aligned}$$

Damit ist gezeigt, dass die average-case Komplexität (unter der speziellen Wahrscheinlichkeitsannahme der Gleichverteilung aller $a[j]$) von BucketSort $\mathcal{O}(n)$ ist.

Nehmen wir an, wir haben m Buckets, bei n zu sortierenden Elementen, wobei m nicht unbedingt gleich n sein muss. Wenn wir obige Rechnung nochmals inspizieren, so erhalten wir in diesem allgemeineren Fall die Komplexität $\mathcal{O}(n(n+m)/m)$. Wenn beispielsweise $m = \sqrt{n}$ gewählt wird, so ergibt sich die Komplexität $\mathcal{O}(n^{3/2})$.

2.7 Selektionsalgorithmen

Die einfachste Selektionsaufgabe besteht darin, das *Maximum* einer Zahlenmenge festzustellen. Sei $V(n)$ die Anzahl der notwendigen Vergleiche (im schlechtesten Fall), um das Maximum von $a[1..n]$ zu bestimmen.

Behauptung:
Für das Bestimmen des Maximums einer n-elementigen Menge gilt $V(n) = n - 1$.

Beweis:

(≤) Gehe nach der üblichen Strategie vor:

Vergleiche $a[1]$ und $a[2]$
Vergleiche $\max(a[1], a[2])$ und $a[3]$
...
Vergleiche $\max(a[1], \ldots, a[n-1])$ und $a[n]$

Dieses sind $n-1$ Vergleiche.

(≥) Da bei jedem Vergleich, den ein Algorithmus tätigt, höchstens ein Element als Nicht-Maximum ausscheiden kann, ergibt sich die Schranke $V(n) \geq n-1$.

□

Tatsächlich zeigt der Beweis auch, dass jeder Entscheidungsbaum (vgl. Abschnitt 2.2) für die Aufgabe, das Maximum zu bestimmen, in jedem Ast mindestens die Länge $n-1$ hat. Daher hat ein solcher Entscheidungsbaum mindestens 2^{n-1} Blätter. Daher gilt auch für den *average-case*: $V(n) = n-1$.

Analoges gilt natürlich auch, wenn das *Minimum* statt das Maximum einer Zahlenmenge bestimmt werden soll.

Als Nächstes wollen wir betrachten, wie viele Vergleiche (im schlimmsten Fall) notwendig sind, um das Minimum *und* das Maximum einer Zahlenmenge zu bestimmen. Sei $V(n)$ die entsprechende Anzahl von Vergleichen.

Man kann natürlich zunächst mit $n-1$ Vergleichen das Minimum bestimmen und dann unter den verbleibenden $n-1$ Elementen mit $n-2$ Vergleichen das Maximum bestimmen, also gilt $V(n) \leq (n-1) + (n-2) = 2n-3$.

Geht es vielleicht noch besser?

Ja, es geht: Man führe zunächst $n/2$ paarweise Vergleiche durch. Danach ergibt sich folgende Situation, die sich durch ein Hasse-Diagramm wie folgt beschreiben lässt:

Sodann bestimme man unter den $\frac{n}{2}$ vielen „Siegern“ mit $\frac{n}{2}-1$ Vergleichen das größte Element und unter den $\frac{n}{2}$ vielen „Verlierern“ mit $\frac{n}{2}-1$ Vergleichen das kleinste Element. Dies macht zusammen $\frac{n}{2} + (\frac{n}{2}-1) + (\frac{n}{2}-1)$ Vergleiche. Also gilt $V(n) \leq \frac{3}{2}n-2$.

Als Nächstes zeigen wir, dass diese obere Schranke optimal ist. Das heißt, wir zeigen, dass *jeder* Algorithmus zum Bestimmen des Maximums und des Minimums *im schlechtesten Fall* mindestens $\frac{3}{2}n-2$ Vergleiche durchführen muss. (Die Aussage *„im schlechtesten Fall“* muss hier extra betont werden, denn bei Vorliegen einer günstigen Eingabepermutation könnte ein Algorithmus bereits mit $n-1$ Vergleichen das Maximum und Minimum bestimmen).

Beweis:
Für das Beweisargument betrachten wir die dynamische Veränderung der folgenden vier Mengen A, B, C, D:

A = Elemente, die bisher bei keinem Vergleich beteiligt waren,

B = Elemente, die bisher bei allen Vergleichen „Sieger“ waren,

C = Elemente, die bisher bei allen Vergleichen „Verlierer“ waren,

D = die restlichen Elemente.

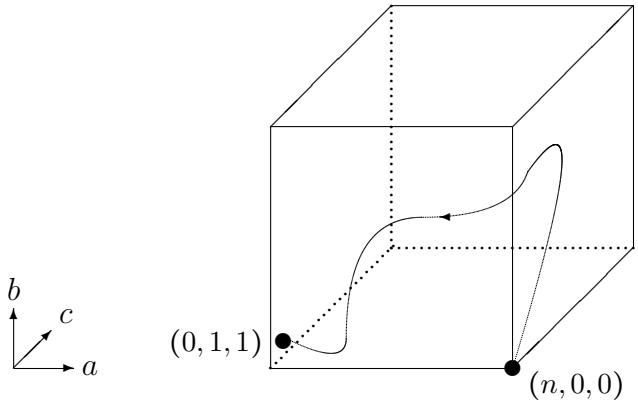
Sei $a = |A|$, $b = |B|$, $c = |C|$, $d = |D|$. Wir beschreiben den „Zustand“, in dem sich ein Algorithmus zur Bestimmung des Minimums und des Maximums befindet, durch (a, b, c). (Es ist nicht notwendig, auch noch d anzugeben, da d durch a, b, c (und n) eindeutig bestimmt ist: $d = n - a - b - c$).
Jeder Algorithmus befindet sich zu Beginn im Zustand $(n, 0, 0)$; und am Ende, nachdem das Maximum und das Minimum eindeutig bestimmt sind, muss der Zustand $(0, 1, 1)$ sein. (Man überlege sich, dass weder $a > 0$ als auch $b, c \neq 1$ möglich sein können). Es geht also darum, in wie vielen Schritten ein Algorithmus (im schlechtesten Fall) den Anfangszustand in den Endzustand überführen kann.

Wir analysieren die möglichen Zustandsübergänge:

$$(a,b,c) \mapsto \begin{cases} (a-2, b+1, c+1), & A < A \\ (a, b-1, c), & B < B \\ (a, b, c-1), & C < C \\ (a, b, c), & D < D \\ (a-1, b, c+1), & A < B \\ (a-1, b, c), & A > B \\ \vdots & \\ \text{(einige weitere Fälle)} & \end{cases}$$

Hierbei bedeutet zum Beispiel „$A > B$“, dass in dem betreffenden Fall ein A-Element und ein B-Element verglichen wurden, und dass das A-Element dabei der „Sieger“ war.
Jeder Algorithmus, der das kleinste und das größte Element einer Zahlenfolge bestimmt, muss daher den Anfangszustand $(n, 0, 0)$ unter Verwendung der oben angegebenen dreidimensionalen „Rösselsprünge“ in den Endzustand $(0, 1, 1)$ überführen.

Skizze:



Da es schwierig ist, sich die möglichen Übergänge von (a, b, c) im dreidimensionalen Raum vorzustellen, bilden wir eine (für das Beweisargument) geeignete Linearkombination von a, b, c und können dann im Eindimensionalen argumentieren. Sei

$$U(a, b, c) = 3a + 2b + 2c$$

der dem Tripel (a, b, c) zugeordnete „Unsicherheitswert“. Dann gilt $U_{\text{Anfang}} = U(n, 0, 0) = 3n$ und $U_{\text{Ende}} = U(0, 1, 1) = 4$. Wenn wir zeigen könnten, dass pro Vergleichsschritt der U-Wert um höchstens 2 abnimmt, so erhalten wir also eine untere Schranke für die Zahl der notwendigen Vergleiche:

$$\frac{U_{\text{Anfang}} - U_{\text{Ende}}}{2} = \frac{3n - 4}{2} = \frac{3}{2}n - 2$$

Dies ist die angekündigte untere Schranke.

Inspizieren wir den ersten Fall oben: Der Wert $U(a, b, c)$ ändert sich hier zu $U(a - 2, b + 1, c + 1)$. Es gilt $U(a, b, c) - U(a - 2, b + 1, c + 1) = 6 - 2 - 2 = 2$. An dieser Stelle wäre unsere Annahme, dass der U-Wert pro Schritt höchstens um 2 abnimmt, also bestätigt.

Der einzige Fall oben, bei dem es nicht so aussieht, ist „$A > B$“. Hier ergibt sich: $U(a, b, c) - U(a - 1, b, c) = 3$. Wir müssen aber bedenken, dass unser Argument sich auf den *schlechtesten Fall* bezieht. Wenn also ein A und ein B Element verglichen werden, so legen wir nun (zur Konstruktion des schlechtesten Falles) fest, dass hier immer das A-Element das *kleinere* sein soll (also nur der Fall „$A < B$“ relevant ist). Für den Fall $A < B$ gilt: $U(a, b, c) - U(a - 1, b, c + 1) = 3 - 2 = 1$, was wieder in Ordnung geht.

Diese Festlegung zur Konstruktion des schlechtesten Falles ist möglich, da es sich bei einem A-Element um ein bisher noch unverglichenes Element handelt, und wir die Wahlfreiheit haben, festzulegen, dass dieses Element zum Beispiel besonders klein sein soll. Wir begeben uns hier also in keinerlei Inkonsistenzen zu vorangegangenen Vergleichsentscheidungen. Wir konstruieren hier also in gewisser Weise dynamisch den schlechtesten Fall für den fraglichen Algorithmus (man spricht oft auch von einem „adversary argument“). Auf ähnliche Weise kann man die „einigen weiteren Fälle“ in der Auflistung oben behandeln. □

Bei der folgenden Selektionsaufgabe wollen wir ausfindig machen, welches das k-kleinste (analog: k-größte) Element ist. Hierbei wird sich zwar herausstellen, welche

$k-1$ Elemente kleiner sind als das gesuchte Element und welche $n-k$ Elemente größer sind. Die Rangordnung dieser restlichen Elemente interessiert uns jedoch nicht.

Man beachte, dass das Problem, den Median einer Menge zu bestimmen, sich als der Spezialfall $k = n/2$ der vorliegenden Problemstellung darstellt.

In Anlehnung an QuickSort bietet sich die folgende Lösung an:

Algorithmus 2.8 `select`(a, i, l, r)

Voraussetzung: $1 \le i \le r - l + 1$

//Bestimmt das i-kleinste Element im Arrayabschnitt $a[l..r]$.

```
1:  IF l = r THEN
2:    return a[l]
3:  ELSE
4:    Bestimme ein Pivotelement x in dem Array a[l..r] //(*)
5:    Ordne a so um, dass alle Elemente, die kleiner (größer) als x sind, links (bzw.
      rechts) von x angeordnet werden
      Sei hierbei q der Index, den das Pivotelement x erhält.
6:    IF i = q - l + 1 THEN
7:      return x
8:    ELSE IF i < q - l + 1 THEN
9:      return select(a, i, l, q - 1)
10:   ELSE
11:     return select(a, i - (q - l + 1), q + 1, r)
```

Aufgerufen wird die Prozedur durch *select*$(a, i, 1, n)$. Es ist klar, dass wie bei QuickSort der schlechteste Fall dann eintritt, wenn das Pivotelement jedes Mal ganz am Rand liegt. Dann hat diese Prozedur die Laufzeit $\mathcal{O}(n^2)$.

Es ist noch nicht spezifiziert, wie das Pivotelement in Zeile $(*)$ ausgewählt werden soll. Betrachten wir zunächst die Variante, bei der das Pivotelement x *zufällig* aus $a[l..r]$ ausgewählt wird. Dies ist *random-select*. Für ein Array a der Größe n und für jedes $j \in \{1, \ldots, n\}$ gilt, dass das Pivotelement x mit Wahrscheinlichkeit $1/n$ das j-kleinste Element von a ist. In diesem Fall hat dann das linke Teilarray die Größe $j-1$ und rechte Teilarray die Größe $n-j$. Schlimmstenfalls bezieht sich der rekursive Aufruf von *select* dann auf das größere der beiden Teilarrays. Der Aufteilungsvorgang des Arrays a benötigt $n-1$ Vergleiche mit dem Pivotelement. Sei $V(n)$ die mittlere Anzahl von Vergleichen von *random-select*. Wegen dieser Vorbetrachtungen gilt die rekursive Beziehung

$$\begin{aligned} V(n) &\le (n-1) + \frac{1}{n} \cdot \sum_{j=1}^{n} \max(V(j-1), V(n-j)) \\ &\le (n-1) + \frac{2}{n} \cdot \sum_{j=n/2}^{n-1} V(j) \end{aligned}$$

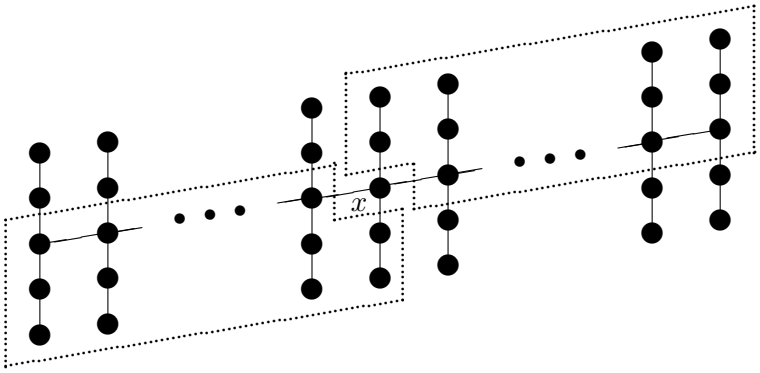
Man prüft leicht per Induktion nach, dass diese Rekursion die Lösung $V(n) \le cn$ für eine genügend große Konstante c hat ($c \ge 4$).

Daher gilt, dass die mittlere Laufzeit des probabilistischen Algorithmus *random-select*, bei beliebiger Eingabe $a[1..n]$ und i, *linear* in der Array-Größe n ist.

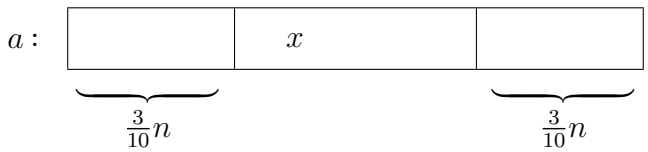
Als Nächstes wollen wir eine *deterministische* Variante von *select* angeben. Hierbei wird die Zeile $(*)$ in kleverer Weise implementiert: Zunächst wird das Array in $n/5$ viele Gruppen mit jeweils 5 Elementen aufgeteilt. In jeder der 5-er Gruppen wird der Median bestimmt (zum Beispiel, indem man jede 5-er Gruppe vollständig sortiert).

Sei $b[1..n/5]$ die Menge dieser $n/5$ vielen Mediane. Wir rufen nun *select* rekursiv auf mit dem Parameter $(b, n/10, 1, n/5)$ (wobei natürlich auf Ganzzahligkeit der Argumente zu achten ist). Hierdurch wird also der Median der $n/5$ vielen Mediane ermittelt. Dieses Element nehmen wir als das Pivotelement x und fahren in dem oben beschriebenen Algorithmus *select* fort.

Das folgende Hasse-Diagramm beschreibt die relative Position der Elemente in den 5-er Gruppen:



In den Bereichen, die durch gestrichelte Linien abgegrenzt sind, befinden sich Elemente, die definitiv kleiner (links unten) bzw. größer (rechts oben) sind als das Pivotelement x, der Median der Mediane. Dieses sind jeweils $\frac{3}{10}n$ viele Elemente. Das bedeutet, dass der Rang von x sich allenfalls noch in dem Intervall $[\,\frac{3}{10}n\,,\,\frac{7}{10}n\,]$ befinden kann:



Schlechtestenfalls kann der rekursive Aufruf von *select* sich auf ein Array der Größe $\frac{7}{10}n$ beziehen. Daher erhalten wir für die Anzahl der Vergleiche von *select* folgende rekursive Abschätzung:

$$V(n) \;\le\; dn/5 + (n-1) + V(n/5) + V(7n/10)$$

Hierbei ist d die Anzahl von Vergleichen, die ausreicht, um den Median in einer 5-er Gruppe zu bestimmen (z.B. $d = 7$). Die $(n-1)$ Vergleiche in der Formel fallen beim Partitionsvorgang an. (Tatsächlich brauchen wir das Pivotelement nur mit denjenigen $\frac{2}{5}n$ vielen Elementen vergleichen, die *nicht* in den durch gestrichelte Linien abgegrenzten Bereichen liegen). Mit Hilfe des Master-Theorems (Seite 17) ergibt sich mit $\alpha_1 = 1/5$, $\alpha_2 = 7/10$, $m = 2$, $k = 1$ die Lösung $V(n) = \Theta(n)$ (wobei der Fall 1 vorliegt). Also hat dieser Selektionsalgorithmus auch im schlechtesten Fall lineare Laufzeit. (Die beste bekannte obere Schranke für das Problem, den Median zu bestimmen (also $k = n/2$), beträgt übrigens $V(n) \le 3n + o(n)$).

Man kann nachweisen, dass jeder Algorithmus zur Bestimmung des Medians im schlechtesten Fall mindestens $V(n) \ge 2n - o(n)$ viele Vergleiche durchführen muss.

Eine etwas schwächere Schranke ergibt sich mit der Beweistechnik von Seite **??** wie folgt.

2.8 Tabellarische Zusammenfassung

Die folgende Tabelle fasst die Ergebnisse dieses Kapitels über die worst-case und average-case Komplexitäten der verschiedenen Sortierverfahren zusammen.

	BubbleSort	MergeSort	QuickSort	HeapSort	BucketSort
worst-case Komplexität	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
average-case Komplexität	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Speicherplatz-bedarf	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$

Als Nächstes sind noch die wesentlichen Ergebnisse über die Anzahl der Vergleiche bei verschiedenen Selektionsaufgaben aufgelistet.

größtes Element	größtes + kleinstes	i-größtes (bzw. Median)
$n-1$	$\frac{3}{2}n-2$	$\Theta(n)$

Kapitel 3

Hashing

Hashing ist eine Methode zur dynamischen Verwaltung von Daten, wobei die Daten durch einen *Schlüssel* angesprochen werden. Viele Anwendungen benötigen nur sehr einfache Daten-Zugriffsmechanismen:

- *Suchen* nach einem Datensatz bei gegebenem Schlüssel;
- Einen neuen Datensatz (samt Schlüssel) *einfügen*;
- Einen Datensatz zu einem gegebenen Schlüssel *löschen*.

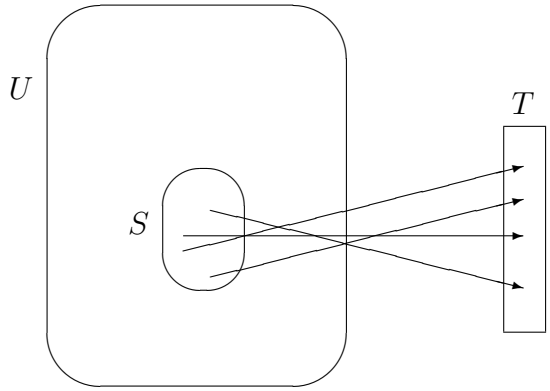
Dieses nennt man oft die *Dictionary Operations*.

Eine mögliche Methode wäre es, die Daten in einem (balancierten) Baum zu verwalten (AVL-Baum, Rot-Schwarz-Baum, B-Baum). In diesem Fall hat jede der obigen Operationen einen (worst-case und average-case) Aufwand von $\Theta(\log n)$. Wir werden sehen, dass Hashing unter gewissen Annahmen einen mittleren Aufwand von $\mathcal{O}(1)$ hat.

Das Szenario bei Hashing besteht darin, dass es eine große Menge U von potenziellen Schlüsseln (das „Universum“) gibt, während die tatsächlich vorkommenden und zu verwaltenden Schlüssel eine relativ kleine Menge $S \subseteq U$ ausmachen. (Im allgemeinen ist aber S vorher nicht bekannt).

Hashing verwendet eine sog. *Hashfunktion* h, welche angewandt auf den (gesuchten oder einzufügenden) Schlüssel $s \in U$ eine Adresse in der *Hashtabelle* liefert. Diese Hashtabelle ist ein Array von Zeigern, welche dann auf die eigentlichen Datensätze zeigen. Im Folgenden nehmen wir an, dass die Hashtabelle T mit $0, 1, \ldots, m-1$ durchindiziert ist.

Nehmen wir für den Moment mal an, dass die Hashfunktion $h : U \longrightarrow \{0, 1, \ldots, m-1\}$ auf der Menge S *injektiv* ist (wobei $|S| \leq m$). Das heißt, je zwei verschiedene tatsächlich verwendete Schlüssel $s, s' \in S$ haben verschiedene Hashwerte $h(s) \neq h(s')$.



In diesem Fall sind die Dictionary Operations sehr einfach in der Zeit $\mathcal{O}(1)$ implementierbar:

- Suchen nach einem Datensatz bei gegebenem Schlüssel:
  IF $T[h(s)] \neq$ `NIL` **THEN** ...
- Einen neuen Datensatz (samt Schlüssel) einfügen:
  $T[h(s)] :=$ (Zeiger auf) Daten zu Schlüssel s
- Einen Datensatz zu einem gegebenen Schlüssel löschen: `DISPOSE`$(T[h(s)])$;
  $T[h(s)] :=$ `NIL`

Diskutieren wir als erstes die Möglichkeiten, eine geeignete Hashfunktion zu definieren. Diese Funktion sollte die Elemente von U möglichst gut in die Menge $\{0, 1, \ldots, m-1\}$ zerstreuen. (Im Deutschen heißt Hashing oft auch *Streuspeicherverfahren*).
Nehmen wir für das Folgende an, die Menge U ist eine Teilmenge der natürlichen Zahlen, $U \subseteq \mathbb{N}$.

- *Divisions-* oder *Kongruenzmethode*:

  $$h(s) := s \text{ MOD } m$$

  Um eine gute Streuung der Werte zu erreichen, empfiehlt es sich m als Primzahl zu wählen.

- *Multiplikationsmethode*: Sei α eine reelle Zahl zwischen 0 und 1.

  $$h(s) := \lfloor m \cdot ((s \cdot \alpha) \text{ MOD } 1) \rfloor$$

  Hierbei bedeutet x MOD 1, dass die Stellen vor dem Komma in x abgeschnitten werden.

  Knuth empfiehlt für α die Zahl $\alpha = (\sqrt{5} - 1)/2 = 0.618...$, eine Zahl, die sich aus dem *goldenen Schnitt* ergibt.

3.1 Das Geburtstagsparadoxon

Im allgemeinen können wir natürlich nicht damit rechnen, dass die Hashfunktion h auf den verwendeten Schlüsseln S injektiv ist. Tatsächlich ist es sogar bei einer sehr kleinen Schlüsselmenge (im Vergleich zu m, der Hashtabellengröße) schon recht wahrscheinlich, dass *Kollisionen* auftreten (also, dass es $s, s' \in S$, $s \neq s'$, gibt mit $h(s) = h(s')$). Dies ist eine Variante des sog. *Geburtstagsparadoxons*: Bei wie vielen (zufällig gewählten) Personen ist es „wahrscheinlich“, dass hiervon zwei am selben Tag (und Monat) Geburtstag haben?

Die zweite Person hat nicht am gleichen Tag Geburtstag wie die erste mit Wahrscheinlichkeit 364/365. Die dritte Person hat nicht am gleichen Tag Geburtstag wie die ersten beiden – unter der Bedingung, dass die ersten beiden bereits verschiedene Geburtstage haben – mit Wahrscheinlichkeit 363/365; usw.

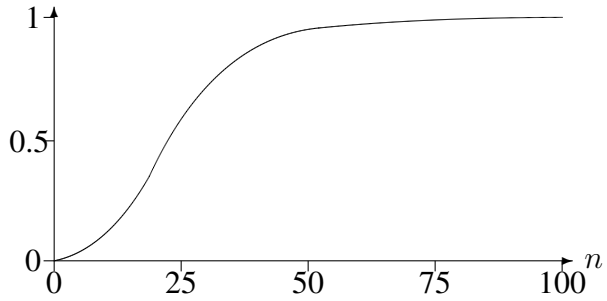
Es stellt sich heraus, dass

$$\frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} \cdots \frac{343}{365} < 0.5$$

Daher ist es ab 23 Personen *wahrscheinlicher*, dass es darunter zwei Personen gibt, die am selben Tag Geburtstag haben, als dass alle 23 Personen an verschiedenen Tagen Geburtstag haben. (Da die Zahl 23 im Vergleich mit 365 außerordentlich klein erscheint, wird dies als ein „Paradoxon“ bezeichnet; eigentlich verbirgt sich hinter dieser Anwendung der Wahrscheinlichkeitsrechnung überhaupt nichts Paradoxes).

Auf Hashing angewandt heißt dies: wenn man (nur!) 23 zufällige Schlüssel in eine Hashtabelle mit $m = 365$ Einträgen einträgt, so ist es „wahrscheinlich“, dass mindestens eine Kollision auftritt.

Das folgende Diagramm zeigt die Wahrscheinlichkeit, dass es bei n Personen zwei gibt, die am selben Tag Geburtstag haben.



Berechnen wir nach dem obigen Muster ganz allgemein, wie groß die Wahrscheinlichkeit ist, dass bei n zufälligen Einträgen in eine Hashtabelle der Größe m alle Hashwerte verschieden sind. Wir können diese Wahrscheinlichkeit wie folgt abschätzen:

$$\prod_{i=1}^{n-1} \frac{m-i}{m} = \prod_{i=1}^{n-1} (1 - i/m) \approx \prod_{i=1}^{n-1} e^{-i/m} = e^{-\sum_{i=1}^{n-1} \frac{i}{m}} = e^{-\frac{n(n-1)}{2m}} \approx e^{-\frac{(n-\frac{1}{2})^2}{2m}}$$

Indem wir diese Wahrscheinlichkeit gleich 0.5 setzen, bestimmen wir den sog. *Median* dieser Wahrscheinlichkeitsverteilung. Bei Vorliegen des Medians gilt somit folgender

Zusammenhang zwischen n und m:

$$n \approx \frac{1}{2} + \sqrt{(2 \ln 2) \cdot m} \approx 0.5 + 1.177 \cdot \sqrt{m}$$

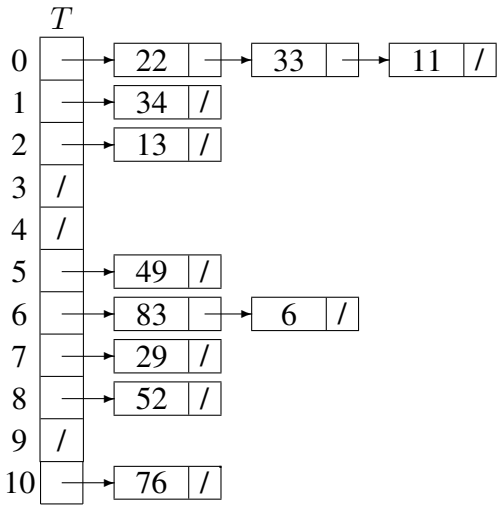
Für $m = 365$ ergibt dies die sehr gute Approximation von $n = 22.99$.

Im Folgenden nennen wir den Quotienten $\beta := n/m$ (n = Anzahl der gespeicherten Elemente; m = Größe der Hashtabelle) den ***Belegungsfaktor*** (oder auch Lastfaktor) der Hashtabelle. Es wird sich bei den Analysen aller zu besprechenden Hashverfahren das erstaunliche Phänomen zeigen, dass der Suchaufwand sich als Funktion von β abschätzen lässt. Das heißt, die Anzahl n der gespeicherten Elemente geht nicht (oder nur indirekt) in den Aufwand ein. Sofern man also einen konstant bleibenden Belegungsfaktor (z.B. $\beta = 0.8$) garantieren kann, ist der Suchaufwand $\Theta(1)$, also unabhängig von n.

3.2 Hashing mit Verkettung

Die vielleicht einfachste Art der Kollisionsbehandlung ist Hashing mit Verkettung. Hierbei wird jedes Element der Hashtabelle als Ausgangspunkt einer (Überlauf)-Liste angesehen. Alle Daten, deren Schlüssel auf denselben Hashwert führen, werden in die entsprechende Liste eingetragen.

Beispiel:
Einzuspeichern seien die Schlüssel $49, 22, 6, 52, 76, 33, 34, 13, 29, 11, 83$. Bei Verwenden der hashfunktion $h(x) = x$ MOD 11 ergeben sich die Hashwerte $5, 0, 6, 8, 10, 0, 1, 2, 7, 0, 6$. Nach Einspeichern in die Hashtabelle ergibt sich die folgende Situation.



In diesem Beispiel gibt es die Kollisionen $h(22) = h(33) = h(11) = 0$ und $h(83) = h(6) = 6$.

Die Dictionary Operations sind leicht implementierbar. Zur Suche nach einem Objekt mit Schlüssel s wird mittels $h(s)$ zunächst der Einstiegspunkt in die entsprechende Liste

berechnet und dann diese Liste linear durchsucht. Analog lässt sich das Einfügen und Löschen implementieren.

Bei der Analyse von Hashverfahren sind generell die folgenden beiden Komplexitätsfunktionen interessant (die als bedingte Erwartungswerte verstanden werden können):

> $A = A(m,n)$ bezeichnet die mittlere Anzahl der Sondierschritte (=Zugriffe auf die Hashtabelle), bis der gesuchte Schlüssel gefunden wird. Hierbei wird vorausgesetzt, dass der Suchschlüssel in der Tabelle tatsächlich vorhanden ist.
>
> Mit $A' = A'(m,n)$ bezeichnen wir dagegen die mittlere Anzahl der Sondierschritte – unter der Bedingung, dass der gesuchte Schlüssel *nicht* vorhanden ist. Das heißt, dies ist die Anzahl der Sondierschritte, die nötig ist, bis das betreffende Hashverfahren sicher sein kann, dass der Schlüssel nicht vorhanden ist.

Das zugrundeliegende Wahrscheinlichkeitsmodell geht davon aus, dass alle Hashtabellenplätze $0, 1, \ldots, m-1$ mit derselben Wahrscheinlichkeit „gehasht“ werden. Insgesamt seien n Schlüssel in der Hashtabelle gespeichert.

Man beachte, dass bei Hashing mit Verkettung durchaus ein Belegungsfaktor $\beta = n/m$ von über 100 % möglich (aber nicht empfehlenswert) ist.

Aus der Analyse von BucketSort (Abschnitt 2.6) ist klar, dass bei Hashing mit Verkettung die mittlere Länge der Überlauflisten gerade β ist. Daher sind im Mittel (bei erfolgloser Suche) gerade $1+\beta$ viele Sondierschritte nötig. Also ist $A' = 1+\beta = 1+n/m$.

Um A zu berechnen, führen wir folgende ganz allgemeine und auch bei anderen Hashverfahren anwendbare Überlegung durch. Die in der Hashtabelle gespeicherten n Schlüssel wurden einmal in irgendeiner Reihenfolge eingetragen. Wir nummerieren die Schlüssel anhand dieser Einfügereihenfolge durch: $s_1, s_2, \ldots, s_n$. Die entscheidende Beobachtung ist, dass die Sondierreihenfolge beim Einfügen von Schlüssel s_i (zum Zeitpunkt $i-1$, also als Schlüssel s_i noch nicht vorhanden war), dieselbe ist wie zum Zeitpunkt n, wenn nach Schlüssel s_i erfolgreich gesucht wird. Wir erhalten also die mittlere, erfolgreiche Suchzeit A, indem wir über die entsprechenden A'-Werte zu allen Zeitpunkten $i = 0, 1, 2, \ldots, n-1$ den Mittelwert bilden:

$$A(m,n) \;=\; \frac{1}{n} \cdot \sum_{i=0}^{n-1} A'(m,i) \qquad (*)$$

Wir setzen $A'(m,n) = 1 + n/m$ in die Formel $(*)$ ein und erhalten:

$$A \;=\; \frac{1}{n} \cdot \sum_{i=0}^{n-1} (1 + i/m) \;=\; 1 + \frac{1}{nm} \cdot \sum_{i=0}^{n-1} i$$

$$= \; 1 + \frac{(n-1)n}{2nm} \;=\; 1 + \frac{\beta}{2} - \frac{1}{2m} \;\leq\; 1 + \frac{\beta}{2}$$

Wir wollen uns ein Bild von einigen statistischen Eigenschaften der Hashtabelle (bei linearer Verkettung der Kollisionselemente) machen. Nehmen wir mal an, es wurde $m = n$ gewählt, also der Belegungsfaktor sei 100 %. Die auf Seite **??** angestellten Überlegungen zeigen, dass nur etwa 63 % der Hashtabellenplätze besetzt (und zum Teil mehrfach besetzt) sind, und dementsprechend etwa 37 % der Plätze frei bleiben.

3.3 Open Hashing

Bei Open Hashing wird im Kollisionsfall ein alternativer freier (bzw. „offener“, daher open) Platz *in* der Hashtabelle gesucht. Es ist klar, dass bei diesem Vorgehen die Zahl der speicherbaren Elemente n höchstens auf m anwachsen kann. Das heißt $\beta = 1$ ist das Maximum. Soll noch ein weiteres Element gespeichert werden, so muss das Verfahren „passen“ – oder man muss nach einer anderen Methode die Daten neu organisieren.

Die einfachste Open Hash-Methode geht mittels *linearem Sondieren* (kurz: *Linear Hashing*). Sollte der Platz $k = h(s)$ beim Einfügen mit einem anderen Schlüssel $s' \neq s$ belegt sein, so wird versucht, den Schlüssel s auf Position $(k + 1)$ MOD m, $(k + 2)$ MOD m, usw. zu speichern, bis eine freie Position gefunden wird.

Umgekehrt, bei der Suche nach einem Schlüssel s in der Hashtabelle wird zunächst die Position $k = h(s)$ sondiert, und falls diese mit einem anderen Schlüssel als der Gesuchte belegt ist, kommt Position $(k + 1)$ MOD m, $(k + 2)$ MOD m, usw. an die Reihe. Dies solange, bis entweder der Schlüssel s gefunden wird, oder bis ein freier Platz angetroffen wird; dann ist der Schlüssel in der Hashtabelle nicht vorhanden.

Ein Problem bei allen Open Hash-Verfahren ist das Löschen. Man kann einen Schlüssel s nicht ohne weiteres einfach aus der Hashtabelle löschen, da dadurch die Sondierreihenfolge für andere Schlüssel unterbrochen werden könnte, und diese dann nicht mehr aufgefunden werden.

Das einfachste Beispiel wären zwei Schlüssel s, s' mit $k := h(s) = h(s')$. Der Schlüssel s sei in Position k gespeichert; der Schlüssel s' sei in Position $k + 1$ gespeichert. Wenn man nun s löscht, so wird eine nachfolgende Suche nach Schlüssel s' fälschlicherweise „nicht vorhanden“ melden, da die Position $h(s') = k$ leer ist.

Man kann dieses Problem behelfsmäßig so lösen, dass man zwischen *gelöschten* und *freien* Plätzen unterscheidet. Bei der Suche nach einem Schlüssel müssen gelöschte Positionen als belegt behandelt werden, und es muss weiter sondiert werden. Erst wenn ein freier Platz gefunden wird, kann „nicht vorhanden“ gemeldet werden.

Bei der Einfügeoperation können gelöschte Plätze dagegen als frei betrachtet werden und es kann dort also der neue Schlüssel eingefügt werden.

Dieses ist jedoch nur eine Notlösung und ist nur dann akzeptabel, wenn nicht sehr häufig Löschoperationen auftreten. Durch die gelöschten Plätze (die beim Suchen als belegt behandelt werden) vergrößert sich die mittlere Suchzeit.

Ein weiteres Problem beim Linear Hashing ist die *Clusterbildung*: Es besteht die Tendenz, dass immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen, sog. *Cluster*. Diese Cluster verschlechtern deutlich die mittlere Suchzeit – im Vergleich zu einer Hashtabelle mit demselben Belegungsfaktor, wo aber die Elemente zufällig gestreut sind.

Wodurch entstehen diese Cluster? Nehmen wir mal an, dass an einer Stelle bereits ein Cluster der Länge 4 vorhanden sei; die nachfolgenden Positionen seien frei.

Die Wahrscheinlichkeit, dass der nächste einzutragende Schlüssel in die Position des linken Pfeiles kommt, ist $5/m$; während die Wahrscheinlichkeit für die rechte Pfeilposition nur $1/m$ ist. Schon bestehende Cluster haben damit – stochastisch gesehen – die Tendenz immer länger zu werden.

Beispiel:
Hier ist das Ergebnis eines Zufallsexperiments. Wir haben eine Hashtabelle mit 100 Einträgen zugrundegelegt und im ersten Experiment diese Tabelle zufällig bis zu einem Belegungsfaktor von 70 % aufgefüllt. Das heißt, wir haben von den $\binom{100}{70}$ vielen 0-1-Folgen mit 70 % Einsen eine Folge zufällig ausgewählt:

Beim zweiten Experiment haben wir mit der Methode Open Hashing mit linearem Sondieren die Tabelle bis zum Belegungsfaktor 70 % gefüllt:

Man kann aufgrund der entstehenden Cluster im zweiten Experiment schon optisch einen Unterschied erkennen.

Wegen der notwendigen Modellierung dieser Clusterbildung ist eine Durchschnittsanalyse von Hashing mit linearem Sondieren nicht ganz einfach. Wir geben zunächst die Formeln für A und A' an (die für großes m und n gelten):

$$A \approx \frac{1}{2} \cdot \left(1 + \frac{1}{1-\beta}\right)$$

$$A' \approx \frac{1}{2} \cdot \left(1 + \frac{1}{(1-\beta)^2}\right)$$

Ein *Rechenbeispiel:* Ein Beamter kann eine Akte pro Tag bearbeiten. Pro Jahr werden zu zufälligen Zeitpunkten insgesamt 300 Akten eingereicht. Wenn der Beamte an 365 Tagen im Jahr arbeitet (zugegeben, ein unrealistisches Beispiel), wie lange bleibt eine Akte im Durchschnitt liegen, bis sie bearbeitet wird? (Es spielt hierbei keine Rolle, ob die Akten in Form eines Stacks oder einer Warteschlange zwischengelagert werden). Die Antwort erhalten wir durch Einsetzen von $m = 365$ und $n = 300$ (also $\beta = 300/365$) in obige Formel für A': Es sind 16.3 Tage.

Die folgende Tabelle listet einige Werte von A und A' auf:

β	A	A'
0.5	1.5	2.5
0.6	1.75	3.62
0.7	2.16	6.05
0.8	3	13
0.9	5.5	50.5
0.95	10.5	200.5

Die Methode *Double Hashing* umgeht das Problem der Clusterbildung durch Verwenden einer zweiten Hashfunktion h'. Diese bestimmt im Kollisionsfall für den jeweiligen Schlüssel s die Schrittweite, mittels der dann in der Hashtabelle weitersondiert werden soll.

Das heißt, die anzuwendende Sondierreihenfolge ist die Folgende:

$$\begin{aligned} h_1(s) &= h(s) \\ h_2(s) &= (h(s) + h'(s)) \text{ MOD } m \\ h_3(s) &= (h(s) + 2h'(s)) \text{ MOD } m \\ h_4(s) &= (h(s) + 3h'(s)) \text{ MOD } m \\ &\vdots \end{aligned}$$

Lineares Sondieren ist dann der Spezialfall $h'(s) = 1$. Die Hashfunktion h' sollte möglichst „unabhängig“ von der Funktion h definiert werden (etwa h mittels Multiplikationsmethode und h' mittels Divisionsmethode). Sollten also zwei Schlüssel s, s' im ersten Versuch kollidieren (das heißt $h(s) = h(s')$), so sollten die jeweiligen Schrittweiten möglichst unterschiedlich sein.

Ferner ist bei diesem Verfahren wichtig, dass die Tabellengröße eine *Primzahl* ist. Denn, wenn m und $h'(s)$ nicht teilerfremd sind, so wird beim Sondieren nicht die gesamte Tabelle (in irgendeiner Reihenfolge) durchlaufen.

Beispiel:
Bei $m = 100$ und $h'(s) = 20$ werden nur 5 verschiedene alternative Tabellenplätze durchlaufen).

Für Primzahlen m gilt jedoch, dass $(\mathbb{Z}_m, +_{\text{mod } m})$ eine zyklische Gruppe ist und jedes Element aus $\mathbb{Z}_m - \{0\}$ ein Generator für diese Gruppe ist. Mit anderen Worten, wenn m eine Primzahl ist, so durchläuft die Sondierreihenfolge *alle* Werte in $\{0, 1, \ldots, m - 1\}$ (in irgendeiner Reihenfolge), bevor sich ein Wert (nämlich $h(s)$) zum ersten Mal wiederholt.

Desweiteren darf der Wert von $h'(s)$ – im Unterschied zu $h(s)$ – nicht Null sein. Die Grund sind offensichtlich. Der Wertebereich für h' ist also $\{1, 2, \ldots, m - 1\}$.

Für Double Hashing gilt:

$$\begin{aligned} A &\approx \frac{1}{\beta} \cdot \ln(\frac{1}{1-\beta}) \\ A' &\approx \frac{1}{1-\beta} \end{aligned}$$

Die folgende Tabelle listet ein paar Werte auf:

β	A	A'
0.5	1.39	2.0
0.6	1.53	2.5
0.7	1.72	3.3
0.8	2.01	5.0
0.9	2.56	10.0
0.95	3.15	20.0

Begründung:
für obige Formeln: $1-\beta$ ist die Wahrscheinlichkeit, auf eine leere Position zu treffen. Falls das gesuchte Objekt nicht vorhanden ist, so ist mit Wahrscheinlichkeit $1-\beta$ nur ein Versuch nötig. Die Wahrscheinlichkeit für 2 Versuche ist für große n näherungsweise $\beta(1-\beta)$, und mit Wahrscheinlichkeit $\beta^2(1-\beta)$ sind 3 Versuche nötig, usw. (Wir nehmen hier an, dass jeder neue Sondierschritt einen zufälligen Tabellenplatz trifft; diese Annahme ist bei Double Hashing gerechtfertigt). Somit gilt:

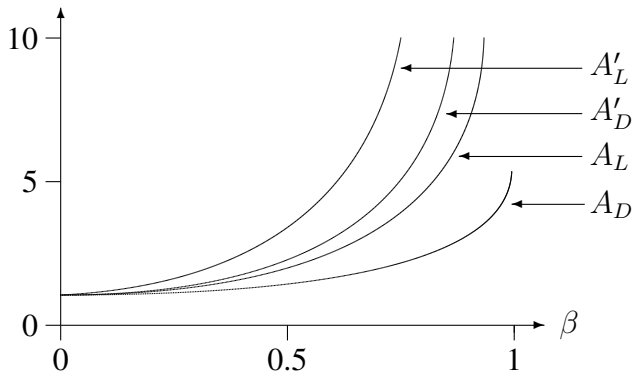
$$\begin{aligned} A' &\approx \sum_{n\geq 1} n\beta^{n-1}(1-\beta) = \frac{1-\beta}{\beta}\sum_{n\geq 1} n\beta^n \\ &= \frac{1-\beta}{\beta}\cdot\frac{\beta}{(1-\beta)^2} = \frac{1}{1-\beta} \end{aligned}$$

Für die Analyse von A verwenden wir die auf Seite 47 entwickelte Formel $(*)$, die auch in diesem Fall anwendbar ist. Damit erhalten wir:

$$\begin{aligned} A &\approx \frac{1}{n}\cdot\sum_{j=0}^{n-1}\frac{1}{1-j/m} = \frac{m}{n}\cdot\sum_{j=0}^{n-1}\frac{1}{m-j} \\ &= \frac{1}{\beta}\cdot(H_m - H_{m-n}) \approx \frac{1}{\beta}\cdot(\ln m - \ln(m-n)) \\ &= \frac{1}{\beta}\cdot\ln\left(\frac{m}{m-n}\right) = \frac{1}{\beta}\cdot\ln\left(\frac{1}{1-\beta}\right) \end{aligned}$$

Hierbei wurde näherungsweise H_n mit $\ln n$ gleichgesetzt. □

Das folgende Diagramm zeigt nochmals zusammenfassend das Verhalten von Double Hashing (A_D und A'_D) und Hashing mit linearem Sondieren (A_L und A'_L).



3.4 Bloomfilter

Ein Bloomfilter ist eine Datenstruktur, mit der effizient festgestellt werden kann, ob ein Element bereits gespeichert bzw. besucht wurde, oder nicht. Dies wird mithilfe von Hashing realisiert. Es wird ein Boolesches Array $A[0..m-1]$ verwendet, dessen Einträge

alle auf `FALSE` initialisiert werden. Außerdem werden k unabhängige Hashfunktionen $h_1, \ldots, h_k$ verwendet. Um Elemente zu speichern wird folgende Methode aufgerufen:

Algorithmus 3.1 `bloomSave(x)`

```
1: FOR i = 1 TO k DO
2:     A[h_i(x)] =TRUE
```

Um zu erfahren ob ein Element x bereits gespeichert wurde, kann man folgende Methode verwenden:

Algorithmus 3.2 `bloomCheck(x)`

```
1: FOR i = 1 TO k DO
2:     IF A[h_i(x)]=FALSE THEN
3:         return FALSE
4: return TRUE
```

Leider besteht die Gefahr von sog. *false positives*, was bedeutet, dass bei `bloomCheck` `TRUE` ausgegeben wird, obwohl das Element nie gespeichert wurde. Sei n die Anzahl gespeicherter Elemente, $m = |A|$ die Größe des Arrays und k die Anzahl der gewählten Hashfunktionen, dann ist der Belegungsfaktor $\beta = \frac{n}{m}$. Somit gilt:

$$P(A[i] = \texttt{FALSE}) = \left(1 - \frac{1}{m}\right)^{n \cdot k} \approx e^{-\frac{nk}{m}} = e^{-k\beta} = p$$

da $1 - x \approx e^{-x}$ für kleine x gilt. Daraus folgt, dass die erwartete Anzahl auf `FALSE` gesetzter Einträge $p \cdot m$ ist. Demnach ist $P(\text{false positive}) = (1-p)^k$. Diesen Ausdruck möchte man möglichst gering halten. Das Minimum dieser Funktion befindet sich bei $k = (\ln 2) \cdot \frac{1}{\beta}$, wobei $p \approx \frac{1}{2}$.

Beispiel:
Wähle $m = 8n$, $\beta = \frac{1}{8}$, $k = 5 \implies p \approx e^{-\frac{5}{8}} = 0.53526$

$$\implies P(\text{false positive}) = (1 - 0.53526)^5 = 0.46474^5 = 0.0216 = 2.16\%$$

Kapitel 4

Dynamisches Programmieren

Im Gegensatz zu divide-and-conquer, welches eine *top-down*-Methode ist, ist dynamisches Programmieren eine *bottom-up*-Methode. Das heißt, um für ein Problem der Größe n eine Lösung zu finden, werden alle für das Problem relevanten Teilprobleme der Größe $1, 2, \ldots, n-1$ gelöst und diese Lösungen in einer Tabelle untergebracht. Um jeweils eine nächstgrößere Teillösung berechnen zu können, muss auf die bisher berechneten Tabelleneinträge zugegriffen werden.

Dynamisches Programmieren wird typischerweise zur Lösung von Optimierungsproblemen angewandt. Die Bezeichnung „Programmierung“ ist historisch bedingt und bezeichnet ein Verfahren, das mit einer Tabelle arbeitet und diese systematisch ausfüllt.

Eine wichtige Voraussetzung zur Anwendung des Verfahrens ist, dass die optimale Lösung für ein Problem der Größe n im Inneren aus optimalen Teillösungen kleinerer Größe zusammengesetzt ist (*Bellmannsches Optimalitätsprinzip*), denn die optimalen Teillösungen sind ja gerade diejenigen, die in der Tabelle erfasst werden und auf die der Algorithmus zurückgreift.

Weiterhin ist dynamisches Programmieren vor allem dann effizient einsetzbar, wenn im Spektrum der verschiedenen Teillösungen viele davon „überlappend“ sind. Das hat die Konsequenz, dass die optimale Lösung eines Teilproblems, die einmal berechnet und in der Tabelle fixiert ist, im Rahmen der Berechnung verschiedener größerer Teilprobleme immer wieder verwendet werden kann, und nicht wieder neu berechnet werden muss. Im Unterschied dazu müsste ein Algorithmus nach dem divide-and-conquer Prinzip diese Teillösungen (und deren Teillösungen) in vielen rekursiven Prozedurinkarnationen immer wieder neu berechnen.

Die Entwicklung eines Algorithmus, der auf dem Prinzip der dynamischen Programmierung beruht, erfolgt in mehreren Schritten:

1. Charakterisiere den Lösungsraum und die Struktur einer erwünschten optimalen Lösung.

2. Definiere rekursiv, wie sich eine optimale Lösung (und der ihr zugeordnete Wert) aus kleineren optimalen Lösungen (und deren Werten) zusammensetzt.

3. Konzipiere den Algorithmus in einer bottom-up Weise so, dass für $n = 1, 2, 3, \ldots$ tabellarisch optimale Teillösungen (und deren zugeordnete Werte) gefunden wer-

den. Beim Finden einer bestimmten optimalen Teillösung der Größe $k > 1$ hilft hierbei, dass bereits alle optimalen Teillösungen der Größe $< k$ bereitstehen.

4.1 Matrizen-Kettenmultiplikation

Gegeben seien n Matrizen, die miteinander multipliziert werden sollen. Diese Matrizen haben unterschiedliche Seitenlängen: die erste ist eine $p_0 \times p_1$ Matrix, die zweite eine $p_1 \times p_2$ Matrix, usw. die letzte ist eine $p_{n-1} \times p_n$ Matrix. Da die Matrizenmultiplikation assoziativ ist, erhebt sich die Frage, in welcher Weise diese Folge von Matrizen am günstigsten zu klammern ist, und dementsprechend, welches die günstigste Auswertungsreihenfolge ist, die mit der geringsten Komplexität auskommt.

Wir beobachten zunächst, dass die Multiplikation von *zwei* Matrizen, wobei die eine eine $p \times q$, die andere $q \times r$ Matrix ist, $p \cdot q \cdot r$ Einzel-Multiplikationen erfordert, und damit einen Gesamtaufwand von $\Theta(p \cdot q \cdot r)$ hat.

Sollen zum Beispiel drei Matrizen M_1, M_2, M_3 multipliziert werden, und diese haben die Seitenlängen (50×10), (10×20), (20×5), so gibt es zwei mögliche Klammerungen und damit zwei mögliche Auswertungsstrategien mit demselben Ergebnis:

$$(M_1 \cdot M_2) \cdot M_3 \text{ und } M_1 \cdot (M_2 \cdot M_3)$$

Die erste erfordert

$$50 \cdot 10 \cdot 20 + 50 \cdot 20 \cdot 5 = 15000$$

Multiplikationen zur Auswertung. Die zweite Strategie erfordert dagegen

$$10 \cdot 20 \cdot 5 + 50 \cdot 10 \cdot 5 = 3500$$

Multiplikationen.

Bei der Multiplikation von n Matrizen können durch günstige bzw. ungünstige Klammerung weit extremere Unterschiede in der Komplexität herauskommen.

Der naive Algorithmus würde alle möglichen Klammerungen ausprobieren, was jedoch exponentiell viele sind.

Ein geschickterer Ansatz ist folgender: Die Problem-Eingabe seien die Matrizen-Seitenlängen $(p_0, p_1, \ldots, p_n)$. Die eigentlichen Matrizen M_i sind für die anvisierte Lösung nicht von Interesse, nur die Dimensionen der Matrizen. (Es geht hier also nicht um das eigentliche Multiplizieren der Matrizen (siehe jedoch Abschnitt 7.2), sondern darum, die optimale Klammerung zu finden, welche die Anzahl der Operationen minimiert).

Sei $m(i, j)$ die minimale Anzahl von Einzel-Multiplikationen, so dass man (bei optimaler Klammerung) den Abschnitt $M_i \cdots M_j$ (gegeben durch die Seitenlängen $(p_{i-1}, p_i, \ldots, p_j)$) berechnen kann.

Es ist klar, dass $m(i, j)$ mit $i = j$ den Wert Null hat.

Nehmen wir an, die optimale (äußere) Klammerung von $M_i \cdots M_j$ sei $(M_i \cdots M_k) \cdot (M_{k+1} \cdots M_j)$ für ein $k \in \{i, \ldots, j-1\}$. Für dieses k gilt dann aufgrund des Optimalitätsprinzips:

$$m(i, j) \;=\; m(i, k) + m(k+1, j) + p_{i-1} \cdot p_k \cdot p_j$$

Der Anteil von $p_{i-1} \cdot p_k \cdot p_j$ kommt durch die Multiplikation der $p_{i-1} \times p_k$ Matrix $(M_i \cdots M_k)$ mit der $p_k \times p_j$ Matrix $(M_{k+1} \cdots M_j)$ zu Stande.
Damit haben wir auch schon eine Rekursionsformel gefunden, die wir zum Ausfüllen der Tabelle verwenden können:

$$m(i,j) \;=\; \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} (m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j), & i < j \end{cases}$$

Der gesuchte Algorithmus verwendet also ein Array $m[1..n, 1..n]$, welches allerdings nur in der Form einer Dreiecksmatrix, also für Indizes (i,j) mit $1 \le i \le j \le n$ verwendet wird. Der Algorithmus trägt alle entsprechenden Tabelleneinträge (i,j) in der Reihenfolge $j - i = 0, 1, 2, \ldots, n-1$ ein, indem er dabei gemäß obiger Formel bereits eingetragene Tabellenwerte verwendet.

Algorithmus 4.1 `dynMatrixMult`$(p_0, \ldots, p_n)$

```
 1: FOR i = 1 TO n DO
 2:    m[i,i] = 0 //Diagonale nullen
 3: FOR l = 1 TO n - 1 DO
 4:    FOR i = 1 TO n - l DO
 5:       j = i + l
 6:       m[i,j] = ∞
 7:       FOR k = i TO j - 1 DO
 8:          q = m[i,k] + m[k+1,j] + p_{i-1} · p_k · p_j
 9:          IF q < m[i,j] THEN
10:             m[i,j] = q
```

Am Ende erfährt man im Array-Element $m[1,n]$ mit wie vielen Einzel-Operationen man bei optimaler Klammerung auskommt. Die optimale Klammerung selbst wird durch den Algorithmus zunächst nicht bestimmt. Diese Information kann man aber leicht dadurch bekommen, dass man sich im Algorithmus merkt (z.B. in einem separaten Array), durch welches k jeweils das Minimum zu Stande kam.

Beispiel:
Gegeben seien die Matrizen $M_1, \ldots, M_6$ mit den Dimensionen $(6, 12, 20, 3, 10, 5, 18)$. Das heißt im einzelnen:

Matrix	M_1	M_2	M_3	M_4	M_5	M_6
Dimension	6×12	12×20	20×3	3×10	10×5	5×18

Der Algorithmus baut das folgende Array m auf.

i \ j	1	2	3	4	5	6
1	0	1440	936	1116	1176	1680
2		0	720	1080	1050	1788
3			0	600	450	1500
4				0	150	420
5					0	900
6						0

Die optimale Klammerung ergibt bei diesem Beispiel eine Gesamtanzahl von Einzel-Multiplikationen von $m[1,6] = 1680$. Die zugehörige optimale Klammerung ergibt sich daraus, wie jeweils die Minima zu Stande gekommen sind. Diese ist

$$(M_1 \cdot (M_2 \cdot M_3)) \cdot ((M_4 \cdot M_5) \cdot M_6)$$

Diese kann man, wie oben angedeutet, dadurch algorithmisch bestimmen, dass man ein zweites Array $Kl[1..n, 1..n]$ verwendet, und im THEN-Teil des obigen Programms die Anweisung $Kl[i,j] := k$ hinzu fügt. In unserem Beispiel ergibt sich die folgende Tabelle:

i \ j	1	2	3	4	5	6
1	-	1	1	3	3	3
2		-	2	3	3	3
3			-	3	3	3
4				-	4	5
5					-	5
6						-

Was die Komplexität von dynamischem Programmieren betrifft, so lässt sich diese abschätzen durch

$$\mathcal{O}\Big(\text{(Größe der Tabelle)} \cdot \text{(Aufwand pro Tabelleneintrag)} \Big)$$

Im Beispiel Matrizen-Kettenmultiplikation ergibt sich somit die Komplexität $\mathcal{O}(n^2) \cdot \mathcal{O}(n) = \mathcal{O}(n^3)$.

Im Prinzip lässt sich die Aufgabenstellung der Matrizen-Kettenmultiplikation genauso auch durch einen divide-and-conquer Ansatz lösen. Schauen wir uns mal die direkte Umsetzung in einen divide-and-conquer Algorithmus an. Dieser Algorithmus benötigt zwar keinen Speicherplatz für eine Tabelle, dafür ist er aber hoffnungslos ineffizient.

Algorithmus 4.2 `d&cMatrixMult`(i, j)

```
1: IF i = j THEN
2:    return 0
3: ELSE
4:    r = ∞
5:    FOR k = i TO j − 1 DO
6:       q = d&cMatrixMult(i, k) + d&cMatrixMult(k + 1, j) + p_{i−1} · p_k · p_j
7:       IF q < r THEN
8:          r = q
9:    return r
```

Aufgerufen wird diese Prozedur dann durch `d&cMatrixMult`$(1, n)$.

Da ein Aufruf von `d&cMatrixMult`(i, j) mindestens 2 weitere rekursive Aufrufe hervorruft (sofern $j - i \geq 2$), hat dieser Algorithmus exponentielle Komplexität.

Das Problem ist, dass in den verschiedenen Verästelungen der Rekursion immer wieder dieselben Werte berechnet werden müssen; `d&cMatrixMult` wird also immer wieder mit denselben Parametern aufgerufen.

Man kann das Problem allerdings mittels *Memorieren* von schon einmal gelösten Teilproblemen beseitigen. Jetzt benötigen wir wie beim dynamischen Programmieren doch wieder eine Tabelle. Wir nehmen an, dass alle Tabelleneinträge mit dem Wert -1 initialisiert sind. Ein (-1)-Wert bedeutet, dass der entsprechende Tabelleneintrag noch unbekannt ist, also noch nicht berechnet wurde.

Algorithmus 4.3 `mixMatrixMult`(i, j)

```
1: IF m[i, j] ≥ 0 THEN
2:    return m[i, j]
3: ELSE IF i = j THEN
4:    m[i, j] = 0
5:    return 0
6: ELSE
7:    r = ∞
8:    FOR k = i TO j − 1 DO
9:       q = mixMatrixMult(i, k) + mixMatrixMult(k + 1, j) + p_{i−1} · p_k · p_j
10:      IF q < r THEN
11:         r = q
12:   m[i, j] = r
13: return r
```

Dieser Algorithmus füllt genauso wie dynamisches Programmieren die entsprechende Tabelle m aus. Der Unterschied ist nur der, dass die Ablaufsteuerung nicht durch FOR-Schleifen, sondern durch Rekursion vorgenommen wird. Die Komplexität dieses Verfahrens ist daher ebenfalls $\Theta(n^3)$.

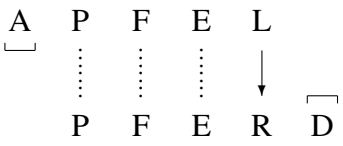
4.2 Editierdistanz

Eine Funktion, die die „Ähnlichkeit“ zweier Strings $a_1 \dots a_m$ und $b_1 \dots b_n$ messen kann, ist die *Editierdistanz*. Dies ist die minimale Anzahl von elementaren Buchstabenoperationen, die notwendig ist, um den einen String in den anderen überzuführen. Unter „elementaren Buchstabenoperationen“ verstehen wir die Folgenden:

- einen neuen Buchstaben einfügen,
- einen Buchstaben löschen,
- einen bestehenden Buchstaben umbenennen.

Jede dieser Operationen schlägt mit dem Wert 1 bei der Editierdistanz zu Buche. (In manchen Anwendungen könnte es sein, dass man die verschiedenen Editieroperationen unterschiedlich gewichtet. Auch das könnte durch den nachfolgenden Algorithmus berücksichtigt werden.) Die maximal-mögliche Editierdistanz von zwei Wörtern der Länge m und n beträgt $\max\{m, n\}$. Der Minimalwert 0 tritt genau dann auf, wenn die beiden Wörter gleich sind.

Beispiel:
Die Editierdistanz zwischen „APFEL“ und „PFERD“ beträgt 3. Die folgende Darstellung zeigt, dass man „A“ löschen, „PFE“ (ohne Editierkosten) übernehmen, „L“ in „R“ umbenennen und „D“ neu einführen kann.



Sei nun $d(i, j)$ die Editierdistanz zwischen den Teilwörtern $a_1 \dots a_i$ und $b_1 \dots b_j$. Es ist klar, dass $d(0, j) = j$ und $d(i, 0) = i$ gilt. Desweiteren erhalten wir folgende rekursive Beziehung (für $i, j > 0$):

$$d(i,j) = \min\left(d(i,j-1)+1,\ d(i-1,j)+1,\ d(i-1,j-1)+\left\{\begin{array}{l} 1,\ a_i \neq b_j \\ 0,\ a_i = b_j \end{array}\right\}\right)$$

Die nachfolgende Tabelle listet die d-Werte auf, die man mit dem folgendem Programm erhält:

Algorithmus 4.4 `editDist`(a, b)

```
 1: FOR j = 0 TO n DO
 2:    d[0, j] = j
 3: FOR i = 0 TO m DO
 4:    d[i, 0] = i
 5: FOR i = 1 TO m DO
 6:    FOR j = 1 TO n DO
 7:       d[i, j] = d[i − 1, j] + 1
 8:       IF d[i, j − 1] + 1 < d[i, j] THEN
 9:          d[i, j] = d[i, j − 1] + 1
10:       IF a[i] = b[j] THEN
11:          k = 0
12:       ELSE
13:          k = 1
14:       IF d[i − 1, j − 1] + k < d[i, j] THEN
15:          d[i, j] = d[i − 1, j − 1] + k
```

L	5	4	3	2	2	3
E	4	3	2	1	2	3
F	3	2	1	2	3	4
P	2	1	2	3	4	5
A	1	1	2	3	4	5
	0	1	2	3	4	5
		P	F	E	R	D

Die Komplexität aller Algorithmen in diesem Abschnitt ist offensichtlich $\mathcal{O}(mn)$.

In gewisser Weise ist die Editierdistanz ein Maß für die Ähnlichkeit der beiden Strings $a_1a_2 \ldots a_m$ und $b_1b_2 \ldots b_n$. Basierend auf diesem Verfahren lassen sich daher auch *approximative String-Matching Algorithmen* entwerfen (vgl. Kapitel 9). Weitere Anwendungen der hier beschriebenen Algorithmen ergeben sich im Kontext der mathematischen Analyse und Rekombination von *DNA-Sequenzen*. Solche Sequenzen können als Strings über dem Alphabet $\{A, T, G, C\}$ (für Adenin, Thymin, Guanin, Cytosin) aufgefasst werden.

4.3 Traveling Salesman Problem

Beim *Traveling Salesman Problem* (kurz: TSP) besteht eine zulässige Eingabe aus einer $n \times n$ Entfernungsmatrix, die die paarweisen Abstände zwischen je zwei Städten i und j angibt ($i, j \in \{1, \ldots, n\}$). Einige Einträge können den Wert ∞ aufweisen, welcher besagt, dass keine direkte Straßenverbindung zwischen Ort i und Ort j existiert. Eine zu einer solchen Eingabematrix zulässige Lösung besteht aus einer Permutation π auf der

Menge $\{1, 2, \ldots, n\}$, welche eine Rundreise über die n Städte repräsentieren soll. Eine zulässige Rundreise darf dabei nicht über einen ∞-Eintrag laufen. Das heißt, keine der Matrixpositionen $(\pi(k), \pi(k+1))$ für $k = 1, \ldots, n-1$ und $(\pi(n), \pi(1))$ darf ∞ sein. Sei $M = (m_{i,j})$ die Eingabematrix. Der Wert c einer solchen zulässigen Permutation π für M ist definiert durch

$$c(\pi) = \sum_{k=1}^{n-1} m_{\pi(k),\pi(k+1)} + m_{\pi(n),\pi(1)}$$

Ohne Beschränkung der Allgemeinheit kann man sich auf solche Permutationen π beschränken mit $\pi(1) = 1$. Man kann die gesuchte Rundreise also immer bei Stadt 1 beginnen lassen. Hierbei geht es darum, eine Lösung mit *minimalem* Wert zu finden; das TSP ist also ein Minimierungsproblem.

Wir lassen in der allgemeinen Formulierung des Problems zu, dass die gegebene Entfernungsmatrix nicht unbedingt symmetrisch zu sein braucht (es muss nicht $m_{ij} = m_{ji}$ gelten); es braucht auch nicht unbedingt die Dreiecksungleichung ($m_{ik} \leq m_{ij} + m_{jk}$) zu gelten. Außerdem dürfen die m_{ij} Einträge auch ∞ sein. Dies heißt dann, dass keine direkte Straße von Stadt i nach Stadt j existiert.

Gesucht ist also nach einer Permutation $\pi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$, die den Wert $c(\pi)$, siehe oben, minimiert.

Wegen der möglichen ∞-Einträge in der Matrix kann es durchaus vorkommen, dass keine Lösung existiert; mit anderen Worten, dass jede Permutation π auf einen unendlich großen $c(\pi)$-Wert führt. Es gibt nur dann eine Lösung, also eine Rundreise, wenn der zugrundeliegende gerichtete Graph (mit $(i,j) \in E \Leftrightarrow m_{i,j} < \infty$) einen Hamilton-Kreis besitzt.

Der naive Algorithmus würde alle $n!$ Permutationen π durchspielen und jedes Mal $c(\pi)$ berechnen. (Es genügt, nur solche Permutationen mit $\pi(1) = 1$ zu betrachten; dieses sind $(n-1)!$ viele). Dieser Algorithmus hätte die Komplexität $\Omega((n-1)!)$, was eine extrem schnell anwachsende Exponentialfunktion ist.

Dieser naive Ansatz kann mittels dynamischen Programmierens verbessert werden. Jedoch wird auch der entsprechende Algorithmus immer noch (gemäßigt) exponentielle Komplexität haben, denn wir wissen ja, dass das TSP ein NP-vollständiges Problem ist, und daher kein Algorithmus mit polynomialer Komplexität (also $\mathcal{O}(n^k)$ für eine Konstante k) zu erwarten ist.

Wenn eine optimale Rundreise (ohne Beschränkung der Allgemeinheit) bei Stadt 1 beginnt und dann die Stadt k besucht, so muss der Weg von k aus durch die Städte $\{2, \ldots, n\} - \{k\}$ zurück nach 1 ebenfalls optimal sein (unter allen solchen Wegen). Daher deutet sich hier wieder das Optimalitätsprinzip an, das man wie folgt umsetzen kann.

Sei $g(i, S)$ die Länge des kürzesten Wegs, der bei Stadt i beginnt, dann durch jede Stadt der Menge S genau einmal geht, um dann bei Stadt 1 zu enden.

Man beachte, dass die Lösung des TSP darin besteht, $g(1, \{2, \ldots, n\})$ zu berechnen. Die Funktion $g(i, S)$ kann wie folgt rekursiv beschrieben werden:

$$g(i, S) = \begin{cases} m_{i1}, & S = \varnothing \\ \min\limits_{j \in S} \Big(m_{ij} + g(j, S - \{j\}) \Big), & S \neq \varnothing \end{cases}$$

Unser dynamischer Programmieralgorithmus benötigt eine Tabelle für die $g(i, S)$-Werte, wobei man allerdings die Kombinationen mit $1 \in S$, als auch die mit $i \in S$ nicht benötigt. Der Algorithmus arbeitet wie folgt:

Algorithmus 4.5 `dynTSP`$(m[n, n])$

```
1: FOR i = 2 TO n DO
2:    g[i, ∅] = m[i, 1]
3: FOR k = 1 TO n − 2 DO
4:    FOR {S : |S| = k, 1 ∉ S} DO
5:       FOR i ∈ {2, . . . , n} − S DO
6:          Berechne g[i, S] gemäß Formel
7: Berechne g[1, {2, . . . , n}] gemäß Formel
```

Die Komplexität des Verfahrens ergibt sich wieder wie üblich

(Größe der Tabelle)·(Aufwand pro Tabelleneintrag)

In diesem Fall ist die Größe der Tabelle (Anzahl der i's)·(Anzahl der S's)$\leq n2^n$. Um einen Tabelleneintrag zu berechnen, muss eine Schleife programmiert werden, die das Minimum unter allen $j \in S$ sucht. Dies ist ein Aufwand von $\mathcal{O}(n)$. Also ergibt dies insgesamt eine Komplexität von $\mathcal{O}(n^2 2^n)$.

Die Funktion $n^2 2^n$ ist zwar exponentiell, jedoch ist $n^2 2^n$ wesentlich kleiner als $n!$ bzw. $(n-1)!$.

Beispiel:
Gegeben sei folgende Entfernungsmatrix:

$$M = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Der Algorithmus berechnet die folgenden Werte:

$$g(2, \varnothing) = m_{21} = 5, \quad g(3, \varnothing) = m_{31} = 6, \quad g(4, \varnothing) = m_{41} = 8$$

$$\begin{aligned} g(2, \{3\}) &= m_{23} + g(3, \varnothing) = 15 \\ g(2, \{4\}) &= m_{24} + g(4, \varnothing) = 18 \\ g(3, \{2\}) &= m_{32} + g(2, \varnothing) = 18 \\ g(3, \{4\}) &= m_{34} + g(4, \varnothing) = 20 \\ g(4, \{2\}) &= m_{42} + g(2, \varnothing) = 13 \\ g(4, \{3\}) &= m_{43} + g(3, \varnothing) = 15 \end{aligned}$$

$$
\begin{aligned}
g(2,\{3,4\}) &= \min\Big(\, m_{23} + g(3,\{4\}),\, m_{24} + g(4,\{3\}) \,\Big) = 25\\
g(3,\{2,4\}) &= \min\Big(\, m_{32} + g(2,\{4\}),\, m_{34} + g(4,\{2\}) \,\Big) = 25\\
g(4,\{2,3\}) &= \min\Big(\, m_{42} + g(2,\{3\}),\, m_{43} + g(3,\{2\}) \,\Big) = 23
\end{aligned}
$$

$$
\begin{aligned}
&g(1,\{2,3,4\})\\
&= \min\Big(\, m_{12} + g(2,\{3,4\}),\, m_{13} + g(3,\{2,4\}),\, m_{14} + g(4,\{2,3\}) \,\Big)\\
&= \min(35, 40, 43)\\
&= 35
\end{aligned}
$$

Indem man nachvollzieht, über welche Minimumsbildungen der gesuchte Wert zu Stande kommt, erhält man auch die zugehörige Lösung. Die optimale Rundreise ist 1–2–4–3–1. (Auf dieses Beispiel wird in Abschnitt 8.2 noch ein anderer Algorithmus angewandt; und weitere algorithmische Lösungen für das TSP finden sich in den Abschnitten 5, 10.3 und 10.5).

4.4 Das 0/1-Rucksackproblem

Die Eingabe für das 0/1-Rucksackproblem besteht aus $2n + 1$ vielen natürlichen Zahlen $v_1, \ldots, v_n$, $g_1, \ldots, g_n$ und G. Die Vorstellung dahinter ist, dass es darum geht, einen Teil der „Objekte“ $1, \ldots, n$ in einen Rucksack der Größe G zu packen; hierbei haben die Objekte die Größen $g_1, \ldots, g_n$. Die einzuhaltende Randbedingung ist, dass die Gesamtgröße der eingepackten Objekte die Rucksackgröße nicht übersteigt. Jedes der Objekte hat einen „Wert“ $v_1, \ldots, v_n$. Es soll eine Auswahl mitzunehmender Objekte solcherart getroffen werden, so dass der Gesamtwert der im Rucksack eingepackten Objekte maximiert wird.

Formal: Gesucht ist ein 0/1-Vektor $(a_1, \ldots, a_n) \in \{0,1\}^n$ mit

$$\sum_{i=1}^{n} a_i g_i \le G \;\text{ und }\; \sum_{i=1}^{n} a_i v_i \to \max$$

Es ist bekannt, dass das 0/1-Rucksackproblem NP-vollständig ist. Einen effizienten Algorithmus werden wir also nicht erwarten können.

Wir nennen das hier betrachtete Problem das 0/1-Rucksackproblem, um es vom Bruchteil-Rucksackproblem zu unterscheiden (das effizient lösbar ist), das später behandelt wird; dort dürfen die Zahlen a_i beliebig aus dem Intervall $[0,1]$ gewählt werden.

Wir beobachten, dass beim 0/1-Rucksackproblem das Optimalitätsprinzip gilt: Wenn der Rucksack der Größe G optimal mit einer Auswahl $I \subseteq \{1, \ldots, n\}$ der Objekte $\{1, \ldots, n\}$ gepackt ist, so gilt für jedes $i \in I$, dass ein $(G - g_i)$-großer Rucksack optimal mit einem Teil der Objekte $\{1, \ldots, n\} - \{i\}$ gepackt ist, indem man die Auswahl $I - \{i\}$ hernimmt. Sei $w(i,h)$ der optimale Wert bei Packen eines Rucksacks der Größe $h \le G$ mit einer

Auswahl der Objekte $1, \ldots, i$, wobei $i \leq n$. Also

$$w(i,h) = \max_{a \in \{0,1\}^i} \Big\{ \sum_{j=1}^{i} a_j v_j \mid \sum_{j=1}^{i} a_j g_j \leq h \Big\}$$

Wegen des Optimalitätsprinzips erhalten wir die Rekursionsgleichung

$$w(i,h) = \begin{cases} 0, & i = 0 \\ w(i-1,h), & i > 0, h < g_i \\ \max\big(w(i-1,h), w(i-1,h-g_i) + v_i \big), & \text{sonst} \end{cases}$$

Die Gleichung erklärt sich wie folgt: Man kann das Objekt i in den Rucksack einpacken oder auch nicht. Wenn man es nicht einpackt, so steht für die Objekte $1, \ldots, i-1$ noch die Größe h zur Verfügung. Wenn man es einpackt, so gewinnt man einen Wert von v_i, allerdings steht für die restlichen Objekte nur noch die Größe $h - g_i$ zur Verfügung (sofern $h - g_i \geq 0$).

Der dynamische Programmieralgorithmus für das 0/1-Rucksackproblem verwendet das Array $w[0..n, 0..G]$ und berechnet die Arraywerte wie folgt:

Algorithmus 4.6 `dyn0/1Rucksack_1`$(v_1, \ldots, v_n, g_1, \ldots, g_n, G)$

```
1: FOR i = 0 TO n DO
2:     FOR h = 0 TO G DO
3:         Berechne w[i,h] gemäß obiger Formel
```

Die optimale Auswahl $a_1, \ldots, a_n$ ergibt sich daraus, welcher Fall bei der Berechnung von $w[n, G]$ jeweils aufgetreten ist.

Die Komplexität des Verfahrens ergibt sich wieder, wie gewohnt, mittels

(Tabellengröße)·(Aufwand pro Tabelleneintrag)

und berechnet sich hier zu $\mathcal{O}(nG) \cdot \mathcal{O}(1) = \mathcal{O}(nG)$. Wenn man als Eingabegröße $m = 2n + 1$ ansieht, da die Eingabe aus $2n + 1$ vielen natürlichen Zahlen besteht, und vom uniformen Komplexitätsmaß ausgeht, dass also jede arithmetische Operation mit $\mathcal{O}(1)$ Kosten verbunden ist, so ist die Komplexität $\mathcal{O}(nG) = \mathcal{O}(m)$, also linear. Wie kann dies angesichts der NP-Vollständigkeit des 0/1-Rucksackproblems sein?

Die Antwort ist, dass es hier wichtig ist, die *Bit-Komplexität* zu betrachten. Wir sprechen hier von einem *pseudo-polynomialen* Algorithmus. Wir müssen also bei den Eingabezahlen, insbesondere was G betrifft, deren Bitlänge, also die Länge der Binärdarstellung als Eingabelänge veranschlagen (vgl. Abschnitt 1.2). Es gilt:

$$\log(G) \approx (\text{Länge der Binärdarstellung von } G) =: k$$

bzw.

$$G \approx 2^k$$

Daher ist die Bit-Komplexität von unserem Algorithmus $\mathcal{O}(n2^k)$, also exponentiell.

Beispiel:
Gegeben sei

$$g_1 = 1, \quad g_2 = 2, \quad g_3 = 3, \quad G = 5$$
$$v_1 = 3, \quad v_2 = 5, \quad v_3 = 6.$$

Der Algorithmus erzeugt die folgende Tabelle:

i \ h	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	3	3	3	3	3
2	0	3	5	8	8	8
3						11

Das Ergebnis ist, dass bei Wahl von Objekt 2 und 3 der Maximalwert von $v_2 + v_3 = 11$ zu erzielen ist (wobei die Rucksackbedingung $g_2 + g_3 \leq G$ eingehalten wird).

Kapitel 5

Greedy-Algorithmen und Matroide

Greedy-Algorithmen sind mit dem dynamischen Programmieren verwandt, jedoch einfacher. Die Grundsituation ist dieselbe: Es geht um ein Optimierungsproblem; es soll sukzessiv eine optimale Lösung – in Bezug auf eine gegebene Bewertungsfunktion konstruiert werden.

Während man beim dynamischen Programmieren solche optimalen Lösungen für alle kleineren Teilprobleme konstruiert und mit Hilfe dieser in einer Tabelle eingetragenen Daten die nächstgrößere optimale Lösung konstruiert, verzichtet man bei Greedy auf die Buchführung mittels einer Tabelle. Stattdessen wird der nächste Erweiterungsschritt zu einer (hoffentlich) optimalen Lösung lediglich aufgrund der lokal verfügbaren Informationen getätigt.

Nehmen wir an, es gibt eine Gewichtsfunktion w, die die „Güte“ einer Lösung (auch einer Teillösung) misst. Es soll eine Lösung mit maximalem w-Wert konstruiert werden. Wir starten mit der leeren Lösung. Schrittweise wird die bisher konstruierte Teillösung erweitert. Wenn zur Erweiterung dieser Teillösung k Erweiterungsmöglichkeiten zur Verfügung stehen, die auf die vergrößerten Teillösungen $l_1, \ldots, l_k$ führen, so wird diejenige Teillösung l_i mit $w(l_i)$ maximal ausgewählt.

Der Name Greedy=gefräßig erklärt sich dadurch, dass ein Greedy-Algorithmus nach der Methode „Nimm immer das größte Stück“ vorgeht.

Dieses einfache Greedy-Prinzip kann in vielen Fällen funktionieren und tatsächlich auf eine optimale Lösung führen – ohne den Aufwand, der bei dynamischem Programmieren betrieben werden muss.

5.1 Bruchteil-Rucksackproblem

Der Unterschied zwischen Greedy und dynamischem Programmieren wird deutlich bei zwei Varianten des Rucksackproblems. Zum einen haben wir das 0/1-Rucksackproblem, das in Abschnitt 4.4 besprochen wurde. Dort wurde eine Lösung mittels dynamischen Programmierens vorgestellt. Zum anderen kann man das *Bruchteil-Rucksackproblem* be-

trachten. Gegeben sind wieder $2n + 1$ Zahlen $v_1, \ldots, v_n$, $g_1, \ldots, g_n$ und G. Die Objekte $1, \ldots, n$, die in den Rucksack der Größe G zu packen sind, dürfen bei dieser Variante in Bruchteilen mitgenommen werden. Eine Lösung besteht nun aus Zahlen $a_1, \ldots, a_n$ mit $0 \le a_i \le 1$, so dass $\sum_{i=1}^{n} a_i g_i \le G$. Gesucht ist eine Lösung, die den Wert $\sum_{i=1}^{n} a_i v_i$ maximiert.

Das Bruchteil-Rucksackproblem (im Vergleich zum 0/1-Rucksackproblem) liefert ein Beispiel für den Begriff *Relaxation*. Das bedeutet, dass die Bedingungen der ursprünglichen Aufgabenstellung (das 0/1-Rucksackproblem) auf eine gewisse Weise „aufgeweicht“ werden, so dass eine effiziente Lösung für das relaxierte Problem (das Bruchteil-Rucksackproblem) möglich wird. Diese Relaxation spielt auch noch eine Rolle bei Branch-and-Bound Algorithmen (Abschnitt 8.1) und Randomized Rounding (Abschnitt 10.1).

Hier geht der Greedy-Ansatz in der folgenden Art und Weise vor: Sortiere alle Objekte gemäß ihrem relativen Wert pro Gewichtseinheit, also nach v_i/g_i. Seien die Objekte so durchnummeriert, dass gilt:

$$\frac{v_1}{g_1} \ge \frac{v_2}{g_2} \ge \cdots \ge \frac{v_n}{g_n}$$

Algorithmus 5.1 `bruchteilRucksack()`

1: **WHILE** $\sum_{i=1}^{k+1} g_i \le G$ **DO**
2: $\quad k = k + 1$
3: $b = \frac{G - \sum_{i=1}^{k} g_i}{g_{k+1}}$
4: `return` $(a_1, \ldots, a_n) = (\underbrace{1, 1, \ldots, 1}_{k}, b, \underbrace{0, 0, \ldots, 0}_{n-k-1})$

Diese Lösung erreicht den Wert $\sum_{i=1}^{k} v_i + b v_{k+1}$.

Wir begründen, dass die Lösung $(1, \ldots, 1, b, 0, \ldots, 0)$ optimal ist.

Beweis:

Wenn wir eine der ersten k Einsen reduzieren auf einen Wert $\alpha < 1$ und stattdessen entsprechend mit einem der Objekte $j > k$ auffüllen, also den Wert a_j erhöhen, so sieht die Bilanz wie folgt aus. Wir reduzieren den Wert der neuen Lösung um $(1 - \alpha)v_i$, und erhöhen ihn gleichzeitig um den Wert βv_j. Hierbei ergibt sich β durch die Betrachtung der Größen. Es muss gelten: $\beta g_j = (1 - \alpha)g_i$, also $\beta = (1 - \alpha)g_i/g_j$. Der Wert der neuen Lösung unterscheidet sich also vom Wert der Lösung $(1, \ldots, 1, b, 0, \ldots, 0)$ um den Betrag $(1-\alpha)g_i v_j/g_j - (1-\alpha)v_i$. Wegen $v_i/g_i \ge v_j/g_j$ ist dieser Betrag ≤ 0. Die neue Lösung ergibt also keine Verbesserung.

Analog lässt sich argumentieren, dass es keine Verbesserung bringt, wenn man den Wert $a_{k+1} = b$ reduziert und stattdessen einen der Werte a_j $(j > k + 1)$ erhöht.

Jede andere Lösung müsste durch eine oder mehrere Transaktionen wie die eben analysierte aus der Lösung $(1, \ldots, 1, b, 0, \ldots, 0)$ hervorgehen. Keine andere Lösung $(a_1, \ldots, a_n)$ kann also einen besseren Wert $\sum_{i=1}^{n} a_i v_i$ erreichen. □

Wir beobachten, dass der Greedy-Ansatz beim 0/1-Rucksackproblem dagegen nicht funktioniert, also nicht immer auf das Optimum führt. Bei dem Beispiel auf Seite 64 etwa würde Greedy die nicht-optimale Lösung (Objekt 1 und Objekt 2) mit dem Wert $v_1 + v_2 = 8$ liefern.

5.2 Matroide

Wir wollen nun versuchen herauszuarbeiten, was allen Problemstellungen, bei denen der Greedy-Algorithmus zu einer optimalen Lösung führt, gemeinsam ist. Es stellt sich heraus, dass die Greedy-Methode dann optimal arbeitet, wenn die zugrundeliegende algebraische Struktur ein sog. *Matroid* ist.

Sei E eine endliche Menge und sei $\mathcal{U}$ eine Menge von Teilmengen von E. Die algebraische Struktur $(E,\mathcal{U})$ heißt ein *Teilmengensystem*, falls gilt:

1. $\varnothing \in \mathcal{U}$,
2. $A \subseteq B,\ B \in \mathcal{U} \ \Rightarrow\ A \in \mathcal{U}$.

Das zu $(E,\mathcal{U})$ gehörige Optimierungsproblem besteht darin, für eine beliebige Gewichtsfunktion $w : E \to \mathbb{R}$ eine in $\mathcal{U}$ maximale Menge T (bzgl. $\subseteq$) zu finden, deren Gesamtgewicht

$$w(T) = \sum_{e \in T} w(e)$$

maximal ist.

(Genausogut könnte man als alternatives Opimalitätskriterium ansetzen, dass eine maximale Menge T mit *minimalem* Gesamtgewicht gesucht ist.)

Wenn wir fordern, dass der Wertebereich von w die *positiven* reellen Zahlen sein sollen, dann können wir uns die Forderung nach einer (bzgl. $\subseteq$) *maximalen* Menge A ersparen, denn in diesem Fall ist jede Menge $A \in \mathcal{U}$, die $w(A)$ maximiert, automatisch maximal (bzgl. $\subseteq$).

Der einem Teilmengensystem $(E,\mathcal{U})$ und Gewichtsfunktion $w : E \to \mathbb{R}$ zugeordnete *kanonische Greedy-Algorithmus* für diese Aufgabe arbeitet wie folgt:

Algorithmus 5.2 `kanonischerGreedy`$(w(e_1),\ldots,w(e_n))$

1: Ordne die Elemente in $E = \{e_1,\ldots,e_n\}$ nach
   absteigendem Gewicht $w(e_1) \geq \ldots \geq w(e_n)$
2: $T = \varnothing$
3: **FOR** $k = 1$ **TO** n **DO**
4: **IF** $T \cup \{e_k\} \in \mathcal{U}$ **THEN**
5: $T = T \cup \{e_k\}$
6: `return` T

(Soll alternativ eine maximale Menge in $\mathcal{U}$ mit *minimalem* Gewicht gefunden werden, so ordne man zu Beginn die Elemente von E nach *aufsteigendem* Gewicht).

Dieser Algorithmus liefert allerdings nicht immer die optimale Lösung.

Beispiel:
Sei $E = \{e_1, e_2, e_3\}, \mathcal{U} = \{\, \varnothing, \{e_1\}, \{e_2\}, \{e_3\}, \{e_2, e_3\} \,\}$ mit $w(e_1) = 3$, $w(e_2) = w(e_3) = 2$. Dies ist offensichtlich ein Teilmengensystem. Der kanonische Greedy-Algorithmus liefert die Lösung $T = \{e_1\}$ mit $w(T) = 3$, während die optimale Lösung dagegen $T' = \{e_2, e_3\}$ mit $w(T') = 4$ ist.

Wir zeigen: Der kanonische Greedy-Algorithmus liefert (in Bezug auf *jede* mögliche Gewichtsfunktion) genau dann optimale Lösungen, wenn das zugrundeliegende Teilmengensystem $(E,\mathcal{U})$ ein sogenanntes *Matroid* ist. Diese algebraische Struktur verlangt ein zusätzliches Axiom, nämlich die *Austauscheigenschaft*:

$$A, B \in \mathcal{U},\ |A| < |B| \implies \exists x \in B - A : A \cup \{x\} \in \mathcal{U}$$

Das obige Beispiel stellt kein Matroid dar, denn die Austauscheigenschaft ist bei $A = \{e_1\}$ und $B = \{e_2, e_3\}$ verletzt.

Wir beobachten, dass in einem Matroid alle (bzgl. $\subseteq$) maximalen Mengen dieselbe Mächtigkeit haben. Denn, wenn sowohl A als auch B maximale Mengen in $\mathcal{U}$ sind mit $|A| < |B|$, dann kann A wegen der Austauscheigenschaft zu einer echt größeren Menge $A \cup \{x\}$ in $\mathcal{U}$ erweitert werden; also kann A nicht maximal gewesen sein.

Das folgende Teilmengensystem ist ein *Beispiel* für ein Matroid: Sei $E = \{1, 2, \ldots, n\}$, und $\mathcal{U}$ seien alle Teilmengen von E der Mächtigkeit $\leq k$, wobei k eine Zahl $\leq n$ ist. Die Austauscheigenschaft ist klar erfüllt. Außerdem ist in diesem Fall offensichtlich, dass alle maximalen Mengen in $\mathcal{U}$ dieselbe Mächtigkeit, nämlich k, haben.

SATZ:
Sei $(E,\mathcal{U})$ ein Teilmengensystem. Der kanonische Greedy-Algorithmus liefert für das zugehörige Optimierungsproblem (in Bezug auf jede beliebige *Gewichtsfunktion $w : E \to \mathbb{R}$) die optimale Lösung* **genau dann wenn** *$(E,\mathcal{U})$ ein Matroid ist.*

Beweis:

($\Leftarrow$) Angenommen, $(E,\mathcal{U})$ ist ein Matroid. Sei $w : E \to \mathbb{R}$ eine beliebige Gewichtsfunktion. Die Elemente von $E = \{e_1, \ldots, e_n\}$ sind nach absteigenden w-Werten angeordnet:

$$w(e_1) \geq \cdots \geq w(e_n)$$

Sei $T = \{e_{i_1}, \ldots, e_{i_k}\}$ die vom Greedy-Algorithmus gefundene Lösung, wobei $i_1 < i_2 < \cdots < i_k$.

Angenommen, es gibt eine Lösung $T' = \{e_{j_1}, \ldots e_{j_k}\}$ mit $w(T') > w(T)$. Dann muss es einen Index μ geben mit $w(e_{j_\mu}) > w(e_{i_\mu})$. Sei μ der *kleinste* solche Index. Da die e_i's nach w-Wert absteigend sortiert sind, gilt also $j_\mu < i_\mu$.

Wir wenden nun die Austauscheigenschaft an auf

$$A = \{e_{i_1}, \ldots, e_{i_{\mu-1}}\} \text{ und } B = \{e_{j_1}, \ldots, e_{j_\mu}\}$$

Wegen $|A| < |B|$ gibt es ein Element $e_{j_\sigma} \in B - A$, so dass $A \cup \{e_{j_\sigma}\} \in \mathcal{U}$. Wegen $w(e_{j_\sigma}) \geq w(e_{j_\mu}) > w(e_{i_\mu})$, hätte der Greedy-Algorithmus dann aber das Element e_{j_σ} bereits vor e_{i_μ} auswählen müssen. Widerspruch.

($\Rightarrow$) Nehmen wir an, die Austauscheigenschaft gelte nicht. Dann gibt es A und B in $\mathcal{U}$ mit $|A| < |B|$, so dass für alle $b \in B - A$ gilt $A \cup \{b\} \notin \mathcal{U}$. Sei $r = |B|$.

Die Gewichtsfunktion $w : E \to \mathbb{R}$ sei definiert durch

$$w(x) = \begin{cases} r+1, & x \in A, \\ r, & x \in B - A, \\ 0, & \text{sonst} \end{cases}$$

Der Greedy-Algorithmus wählt dann eine Menge T mit $A \subseteq T$ und $T \cap (B - A) = \varnothing$. Also ist $w(T) = (r+1) \cdot |A| \leq (r+1)(r-1) = r^2 - 1$. Wählt man stattdessen eine Lösung $T' \supseteq B$, so hat diese den Wert $w(T') \geq r \cdot |B| = r^2$.

Der Greedy-Algorithmus liefert in diesem Fall also nicht die optimale Lösung. □

Wenn wir also bei einem Problem, bei dem sich der Greedy-Ansatz anbietet, feststellen, dass die zugrundeliegende algebraische Struktur ein Matroid ist, dann haben wir mit Hilfe dieses Satzes „automatisch“ gezeigt, dass Greedy die optimale Lösung liefert.

Hierzu werden wir im Folgenden ein paar Beispiele liefern.

5.3 Aufspannende Bäume: Kruskal-Algorithmus

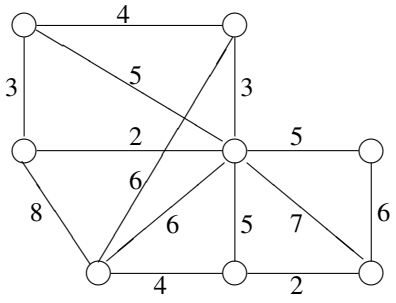
Ein weiteres Beispiel, wo Matroide in natürlicher Weise verwendet werden können, stammt aus dem Bereich der Graphentheorie. Sei $G = (V, E)$ ein gegebener ungerichteter, zusammenhängender Graph. Einem solchen Graphen kann man ein Matroid zuordnen, das wir *Graphen-Matroid* nennen, wie folgt: Die Grundmenge ist die Menge aller Kanten E; und als Teilmengensystem $\mathcal{U}$ über E nehmen wir alle solche Kantenmengen, die keinen Kreis enthalten.

Dieses Teilmengensystem ist tatsächlich ein Matroid, denn seien A und B zwei zyklenfreie Teilmengen von E mit $|A| < |B|$. Sowohl A als auch B zerlegen die zugrundeliegende Knotenmenge V in disjunkte Knotenbereiche: zwei Knoten gehören zum selben Bereich, wenn sie durch einen Weg in A (bzw. in B) miteinander verbunden sind. Sei $V = V_1 \cup V_2 \cup \cdots \cup V_k$ die durch A induzierte Zerlegung. Jede Kante in B verbindet entweder zwei Knoten im selben Bereich V_i oder in zwei verschiedenen Bereichen V_i und V_j. In B können höchstens $\sum_i (|V_i| - 1) = |A|$ viele Kanten vom ersten Typ vorkommen, da B zyklenfrei ist. Da $|B| > |A|$, muss es also mindestens eine Kante in $B - A$ geben, die zwei verschiedene Bereiche verbindet. Eine solche Kante kann zu A hinzugefügt werden, ohne dass ein Zyklus entsteht. Also haben wir es mit einem Matroid zu tun.

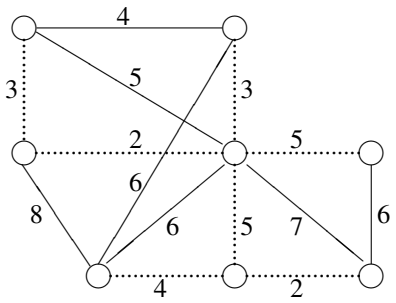
Als Nächstes nehmen wir an, es sei eine Gewichtsfunktion $w : E \to \mathbb{R}$ gegeben, also eine Gewichtung der Kanten des zugrundeliegenden Graphen. Der kanonische Greedy-Algorithmus berechnet eine maximale Menge von Kanten mit maximalem Gewicht (oder minimalem Gewicht – je nachdem, ob wir die Kanten absteigend oder aufsteigend anordnen). Da die Kantenmenge maximal ist, besteht diese nur noch aus *einer* Zusammenhangskomponente, das heißt, das Ergebnis ist ein sog. *aufspannender Baum* des Graphen G. Also ein zusammenhängender Teilgraph, auf dem alle Knoten vorkommen,

und der keinen Zyklus enthält. Dieser Algorithmus (normalerweise in der Variante, dass ein aufspannender Baum mit *minimalem* Gewicht gefunden wird) heißt auch *Kruskal-Algorithmus*.

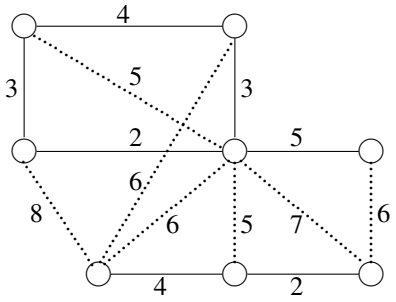
Beispiel:
Gegeben sei folgender kanten-bewertete Graph:



Der Kruskal-Algorithmus wählt nun – nach aufsteigendem Kantengewicht – Kante für Kante aus – solange diese Kante keinen Kreis schließt. Das Ergebnis ist folgender aufspannender Baum mit *minimalem* Kantengewicht, nämlich $2 + 2 + 3 + 3 + 4 + 5 + 5 = 24$ (gestrichelt dargestellt):



Sofern man nach absteigendem Kantengewicht die Kanten auswählt, erhält man einen aufspannenden Baum mit *maximalem* Kantengewicht, nämlich $8+7+6+6+6+5+5 = 43$ (im Folgenden Bild wieder gestrichelt gezeichnet):



5.4 Kürzeste Wege: Dijkstra-Algorithmus

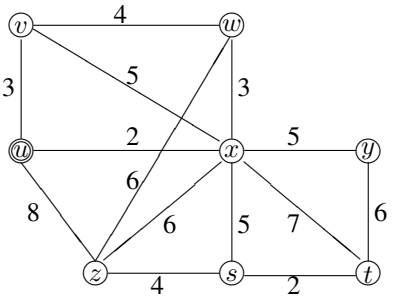
Gegeben sei wieder ein kanten-bewerteter (gerichteter oder ungerichteter), zusammenhängender Graph, also $G = (V, E)$ und $w : E \to \mathbb{R}_+$, wobei ein Knoten $u \in V$ (der „Startknoten“) besonders ausgezeichnet ist. Gesucht sind alle kürzesten Wege von u aus zu

jedem beliebigen anderen Knoten $v \in V$. Der *Algorithmus von Dijkstra* löst dieses Problem wie folgt. Hierbei ist W eine Liste der noch zu sondierenden Knoten (am Anfang ist $W = V$, am Ende ist $W = \varnothing$); F ist eine Auswahl an Kanten, welche die kürzesten Wege von u aus zu allen anderen Knoten ausmachen; $l(v)$ ist die kürzestmögliche Weglänge von u nach v und $k(v)$ ist die optimale zu v führende Kante. Für einen Knoten v bezeichnet $Adj(v) = \{v' \mid (v, v') \in E\}$.

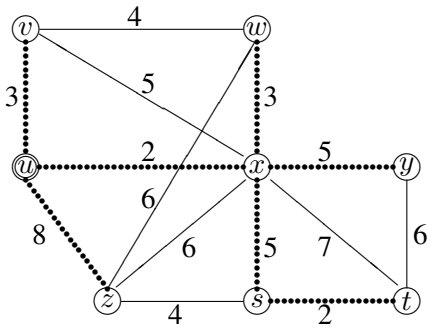
Algorithmus 5.3 `dijkstra`(V, u)

```
 1: FOR v ∈ V DO
 2:    l(v) = ∞
 3: l(u) = 0, W = V; F = ∅
 4: FOR i = 1 TO |V| DO
 5:    Finde einen Knoten v ∈ W mit l(v) minimal
 6:    W = W − {v}
 7:    IF v ≠ u THEN
 8:       F = F ∪ {k(v)}
 9:    FOR v' ∈ Adj(v) ∩ W DO
10:       IF l(v) + w()v, v') < l(v') THEN
11:          l(v') = l(v) + w(v, v')
12:          k(v') = (v, v')
```

Beispiel:
Wir nehmen wieder den obigen Beispielgraphen:



Vom Startknoten u aus entwickelt der Dijkstra-Algorithmus die folgenden (gestrichelten) Pfade (dies sind die Kanten in F). Hierbei werden die Knoten in folgender Reihenfolge aus W entfernt: u, x, v, w, y, s, z, t.



Diese Pfade bilden wieder einen aufspannenden Baum, der allerdings nicht minimales Gewicht hat; was stattdessen minimiert wird, sind die Wegstrecken von u aus gesehen.

Was die Komplexität des Dijkstra-Algorithmus betrifft, so sieht man, dass eine äußere Schleife durchlaufen werden muss; diese liefert den Faktor $\mathcal{O}(|V|)$. Im Inneren dieser Schleife ist aber eine weitere, die für das Auffinden des Minimums zuständig ist (Komplexität $\mathcal{O}(|V|)$). Das Aufsuchen aller Nachbarn von v kann mit $\mathcal{O}(|V|)$ abgeschätzt werden. Daher ist die Komplexität des Dijkstra-Algorithmus beschränkt durch $\mathcal{O}(|V|^2)$. Wir werden allerdings noch Komplexitätsverbesserungen in Abschnitt 5.6 besprechen.

Die Korrektheit des Dijkstra-Algorithmus kann man sich durch Aufzeigen der entsprechenden Matroid-Struktur klarmachen, und somit auf die Korrektheit des kanonischen Greedy-Algorithmus für Matroide zurückführen. Wir wählen dieses Mal als Grundmenge die Menge aller zyklenfreien Pfade vom Startknoten u aus. Das Teilmengensystem $\mathcal{U}$ über dieser Grundmenge besteht aus allen solchen Pfadmengen, die auf *verschiedene* Endknoten führen. Man sieht leicht ein, dass diese Struktur ein Matroid ist, denn wenn A und B Pfadmengen aus $\mathcal{U}$ sind mit $|A| < |B|$, dann gibt es in A genau $|A|$ verschiedene Endknoten und in B befindet sich mindestens ein Pfad mit einem weiteren Endknoten, der in A nicht vorkommt. Daher kann A um diesen Pfad erweitert werden.

Als Nächstes benötigen wir noch eine geeignete Gewichtsfunktion w' auf der Grundmenge, also auf den von u ausgehenden Pfaden. Sei p ein solcher Pfad. Dann setzen wir

$$w'(p) = \sum_{k \text{ liegt auf } p} w(k)$$

Man überzeugt sich leicht davon, dass der kanonische Greedy-Algorithmus für dieses Matroid und diese Gewichtsfunktion w' im Ablauf und im erzeugten Ergebnis exakt mit dem Dijkstra-Algorithmus übereinstimmt. (Der Dijkstra-Algorithmus ist nur effizienter formuliert; man muss nicht alle zyklenfreien Pfade, die von u ausgehen, erzeugen und nach aufsteigenden w'-Werten sortieren, wie dies beim kanonischen Greedy-Algorithmus vorgesehen ist). Im Beispiel oben wählt der kanonische Greedy-Algorithmus der Reihe nach folgende Pfade:

Pfad p	$w'(p)$
u	0
$u-x$	2
$u-v$	3
$u-x-w$	5
$u-x-y$	7
$u-x-s$	7
$u-z$	8
$u-x-s-t$	9

Als Letztes wollen wir noch bemerken, dass man den Dijkstra-Algorithmus durchaus auch als einen einfachen dynamischen-Programmier-Algorithmus ansehen kann, denn es wird die zunächst leere (eindimensionale) Tabelle der l-Werte aufgebaut, die die kürzesten Weglängen angibt. In jedem Erweiterungsschritt wird auf die bereits berechneten

l-Werte zurückgegriffen. Es gilt auch das Bellmannsche Optimalitätsprinzip: Die kürzeste Wegstrecke von u nach v erhält man, indem man denjenigen Vorgänger v' von v auswählt, der den Wert $l(v') + w(\{v', v\})$ minimiert. Dieses Vorgehen entspricht also (auch) dem dynamischen Programmier-Paradigma.

5.5 Optimale Codes: Huffman-Algorithmus

Bei der Huffman-Codierung wird ausgenützt, dass die Zeichen, die in einer Datei vorkommen (insbesondere, wenn es sich um einen deutschen oder englischen Text handelt), typischerweise mit sehr unterschiedlichen Häufigkeiten vorkommen. Statt für jedes Zeichen z.B. 1 Byte (=8 Bit) zu verwenden, werden diesen nun Codewörter unterschiedlicher Länge zugeordnet: Häufig auftretende Buchstaben erhalten kurze Codewörter und seltene Buchstaben erhalten längere Codewörter.

Gegeben sei eine Wahrscheinlichkeitsverteilung auf einem Alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$. Es sei p_i die Wahrscheinlichkeit für das Zeichen a_i. (Es ist für das Folgende unwichtig, ob die p_i's Wahrscheinlichkeiten oder absolute Häufigkeiten sind; die p_i's brauchen also in der Summe nicht 1 zu ergeben).

Gesucht ist nun ein optimaler *Präfixcode* für diese Wahrscheinlichkeits- (oder Häufigkeits-) Verteilung auf Σ. Ein Präfixcode ist eine Abbildung $c : \Sigma \to \{0,1\}^*$, so dass kein Codewort $c(a_i)$ Anfangsstück (Präfix) eines anderen Codeworts $c(a_j)$ ist. Man beachte, dass solche Codes *variable Länge* haben; nicht jedes Zeichen wird mit derselben Anzahl von Bits dargestellt.

Ein Präfixcode heißt *optimal* (in Bezug auf die gegebene Wahrscheinlichkeitsverteilung), wenn er die mittlere Codewortlänge $\sum_{i=1}^{n} p_i \cdot |c(a_i)|$ minimiert.

Beispiel: Sei $\Sigma = \{a, b, c, d, e, f\}$, wobei

$$\begin{array}{llll} p(a) = 0.45 & & p(d) = 0.16 \\ p(b) = 0.13 & & p(e) = 0.09 \\ p(c) = 0.12 & & p(f) = 0.05 \end{array}$$

Ein möglicher Präfixcode wäre

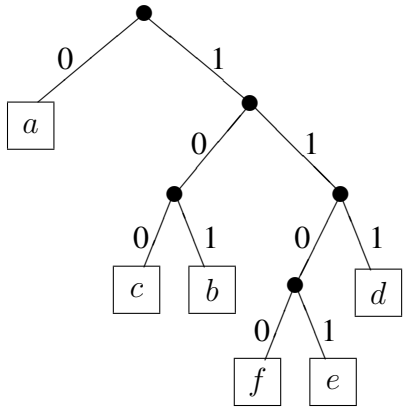
$$\begin{array}{llll} a \mapsto 0 & & d \mapsto 111 \\ b \mapsto 101 & & e \mapsto 1101 \\ c \mapsto 100 & & f \mapsto 1100 \end{array}$$

Dieser Code hat eine mittlere Codewortlänge – bezogen auf die gegebene Wahrscheinlichkeitsverteilung – von:

$$0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4 \; = \; 2.24$$

Tatsächlich ist dieser Präfixcode – in Bezug auf die mittlere Codewortlänge – optimal.

Jeder Code lässt sich in Form eines *Codebaumes* darstellen: Man steigt bei der Wurzel ein; nach links gehen bedeutet 0; nach rechts gehen bedeutet 1. Ein Präfixcode zeichnet sich dadurch aus, dass Zeichen des zu codierenden Alphabets nur an den Blättern anzufinden sind.



Ein Präfixcode hat die Eigenschaft, dass er *eindeutig entzifferbar* ist; das heißt, in einer fortlaufenden Folge von Nullen und Einsen können die Codewörter (deren Beginn und Ende) eindeutig identifiziert werden. Wenn beispielsweise die Folge

$$00110111100110011100 0$$

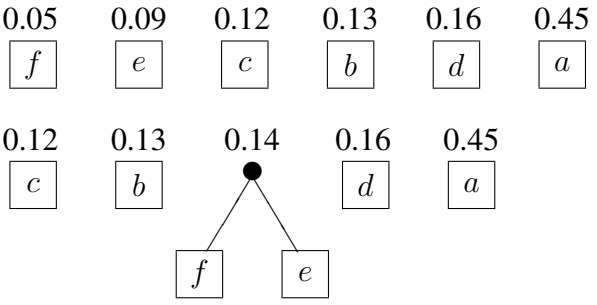
gegeben ist, so ist die Decodierung eindeutig möglich, indem man den Codebaum wie einen endlichen Automaten verwendet; man startet in der Wurzel und jedes Mal wenn man auf ein eckiges Kästchen stößt, hat man ein Codewort erkannt:

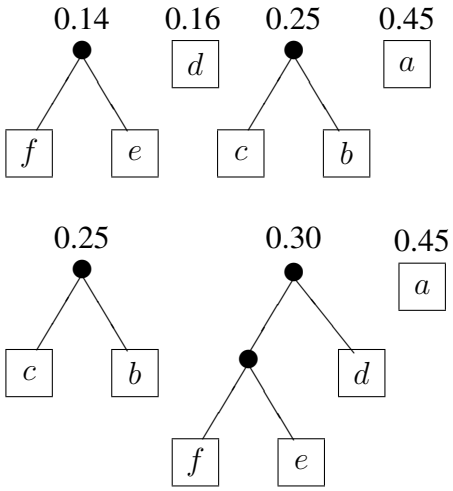
$$\underbrace{0}_{a}\ \underbrace{0}_{a}\ \underbrace{1101}_{e}\ \underbrace{111}_{d}\ \underbrace{0}_{a}\ \underbrace{0}_{a}\ \underbrace{1100}_{f}\ \underbrace{111}_{d}\ \underbrace{0}_{a}\ \underbrace{0}_{a}\ \underbrace{0}_{a}$$

Der Algorithmus nach Huffman konstruiert einen solchen optimalen Präfixcode, bzw. den zugeordneten Codebaum, nach der Greedy-Methode. Zunächst werden alle Symbole $a_1, \ldots, a_n$ als Bäume, die aus jeweils einem Knoten bestehen, aufgefasst. Der jedem solchen Baum zugeordnete *Wert* ist $p(a_i)$. In jedem Schritt werden jeweils *zwei* solche Bäume zusammengefasst zu *einem* Baum, indem man diejenigen mit den *kleinsten* Werten auswählt und eine gemeinsame Wurzel über die beiden Bäume stülpt. Der dem solcherart geschaffenen Baum zugeordnete Wert ist die *Summe* der beiden Werte der Teilbäume.

Bei dem obigen Beispiel würde man im ersten Schritt die beiden Bäume, die den Blättern e und f entsprechen, zusammenfassen. Der Wert des neuen Baumes ist dann $0.09+0.05 = 0.14$. Im nächsten Schritt würde man die Knoten b und c (mit den Werten 0.13 und 0.12) zu einem Baum mit dem Wert 0.25 zusammenfassen, usw. Das Endergebnis des Huffman-Algorithmus ist genau der oben angegebene Codebaum.

Die folgenden Bilder zeigen einige Schritte im Algorithmenablauf.





Diese Bilder suggerieren, dass man die jeweiligen Häufigkeitswerte vollständig sortieren muss, um dann nach Zusammenfassen der zwei am wenigsten häufigen Teilbäume, diese wieder in die sortierte Liste einzufügen. Unter Verwenden von Priority Queues (vgl. Abschnitt 5.6) kann dies aber effizienter implementiert werden. Und zwar wird zunächst der Heap (als Priority Queue) mit den $n = |\Sigma|$ vielen Häufigkeitswerten initialisiert. Dies kann wie bei der ersten Phase von HeapSort (Abschnitt 2.5) mit Komplexität $\mathcal{O}(n)$ geschehen. Dann werden in jedem Schritt die zwei Symbole mit den kleinsten Häufigkeitswerten gesucht und aus dem Heap entfernt (jeweils Komplexität $\mathcal{O}(\log n)$), deren Werte addiert und wieder in den Heap eingefügt (Komplexität $\mathcal{O}(\log n)$). Insgesamt ergibt sich so ein Verfahren der Komplexität $\mathcal{O}(n \log n)$.

Um den Huffman-Algorithmus zur Datenkompression zu verwenden, durchläuft man die Datei in einem ersten Durchlauf, um eine Häufigkeitstabelle für die vorkommenden Zeichen (Bytes) aufzustellen. In einem englischen oder deutschen Text wird der Buchstabe e etwa viel häufiger auftreten als der Buchstabe x. Anhand der Häufigkeitstabelle erstellt man den Huffman-Codebaum. Schließlich speichert man die Datei, indem man zunächst den Codebaum geeignet als Bitstring codiert (oder die Häufigkeitstabelle, die der Konstruktion des Codebaums zugrunde liegt), gefolgt von den Zeichen in der Datei in der neuen Präfixcode-Darstellung. Eine typische (deutsche oder englische) Textdatei kann man so auf etwa 2/3 der ursprünglichen Länge komprimieren. Ein Möglichkeit der Verbesserung besteht darin, nicht nur für Einzelzeichen einen neuen Code vorzusehen, sondern für je zwei (oder drei,...) aufeinander folgende Zeichen. Der Aufwand für die zugehörige Häufigkeitstabelle steigt dann jedoch immens an, denn man benötigt anstatt $|\Sigma|$ nun $|\Sigma|^2$ bzw. $|\Sigma|^3$ viele Einträge. Es lässt sich zeigen, dass der Huffman in dem Sinne optimal ist, dass kein anderer Codebaum eine geringere mittlere Codewortlänge erreicht.

5.6 Priority Queues und Union-Find

Bei der Implementierung von allen Greedy-Algorithmen steht immer wieder die Basisaufgabe an, aus einer Menge von möglichen Objekten e_i ($i = 1, \ldots, n$) dasjenige mit maximalem (oder minimalem) $w(e_i)$-Wert herauszufinden. In unserer bisherigen Algorith-

menformulierung haben wir das immer so gemacht, dass alle Elemente e_i nach absteigenden (oder aufsteigenden) w-Werten vorzusortieren sind (was Komplexität $\mathcal{O}(n \log n)$ hat).

Tatsächlich lässt sich dies effizienter implementieren, wenn eine Datenstruktur zur Verfügung steht, die die folgenden Grundoperationen unterstützt:

- *Initialisiere die Datenstruktur:* das heißt, aus der zu verwaltenden Datenmenge wird die fragliche Datenstruktur hergestellt.
- *Extrahiere das Maximum:* aus der zu verwaltenden Datenmenge soll dasjenige mit maximalem Wert zurückgeliefert und aus der Datenstruktur entfernt werden.
- *Füge ein Element ein:* ein beliebiges neues Datenelement, dem wiederum ein Wert zugeordnet ist, soll in die bestehende Datenstruktur eingefügt werden.

Eine sog. *abstrakte Datenstruktur*, die genau diese Operationen (möglichst effizient) unterstützt, nennt man *priority queue* (auch: Prioritätsschlange oder Vorrangschlange). Man nennt eine solcherart über die gewünschten Operationen gegebene Datenstruktur *abstrakt*, um diese von der *konkreten* Implementierung zu unterscheiden. (Genauso wie der mathematische Begriff der *Gruppe* eine *abstrakte* Struktur darstellt, die lediglich durch die Gruppenaxiome spezifiziert wird. Etwas anderes ist dann eine *konkrete* Gruppe, wie zum Beispiel $(\{0,1\}, \oplus)$).

Man kann die Eigenschaften einer abstrakten Datenstruktur auch axiomatisch beschreiben, worauf wir hier aber verzichten wollen; jeder weiß wohl, was mit „Maximum einer endlichen Menge“ und „Einfügen eines Elements in eine Menge“ gemeint ist. Andere in der Informatik bekannte abstrakte Datenstrukturen sind zum Beispiel der Keller (stack) und die Schlange (queue).

Eine mögliche konkrete Implementierung einer priority queue besteht darin, balancierte Suchbäume (zum Beispiel AVL-Bäume) zu verwenden: Um das Maximum einer in einem AVL-Baum gehaltenen Datenmenge zu bestimmen, starte man bei der Wurzel und gehe bis auf Blattebene herunter, indem man immer den rechten Sohnknoten auswählt. Die Anzahl der Schritte ist durch die Tiefe des AVL-Baums beschränkt; diese ist bei n gespeicherten Daten gerade $\Theta(\log n)$.

Das Herausnehmen eines Elements erfordert evtl. einige Rebalancierungsschritte (Rotationen oder Doppelrotationen), die sich entlang des Pfades bis zur Wurzel vollziehen; auch das Löschen hat somit die Komplexität $\Theta(\log n)$.

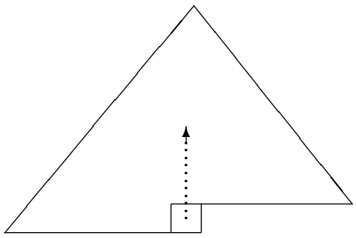
Analoges gilt für das Einfügen eines neuen Elements, man muss die Einfügestelle aufsuchen und dann Rebalancieren; dies hat die Komplexität $\Theta(\log n)$. Um den Initialisierungsprozess durchzuführen, kann man n-mal die Einfügeoperation ausführen; dies hat Komplexität $\mathcal{O}(n \log n)$.

Eine andere, noch interessantere Möglichkeit, die priority queue konkret zu implementieren, ist mittels *Heaps*. Das Maximum der im Heap gespeicherten Datenmenge ist immer an der Wurzel zu finden. Das Entfernen das Maximums geht so wie bei HeapSort beschrieben: man entfernt die Wurzel und setzt stattdessen das „letzte“ Blatt (das in

der rechtesten Position auf tiefster Ebene sitzende Blatt) an die Wurzelposition. Da die Heap-Eigenschaft an der Wurzel dann im allgemeinen verletzt ist, lässt man das Element dann mittels Heapify in den Heap hineinsinken, bis die Heap-Eigenschaft wieder erfüllt ist. Diese Operation ist auf Seite 27ff beschrieben. Die Komplexität dieser Operation ist bei n gespeicherten Elementen wieder $\Theta(\log n)$.

Bei manchen Greedy-Algorithmen muss nicht das Maximum sondern das Minimum aufgesucht werden. Der Heap lässt sich analog natürlich auch so organisieren, dass sich das Minimum an der Wurzel befindet.

Soll ein neues Element in den Heap eingefügt werden, so wird dieses zunächst auf die freie „letzte Blatt"-Position gesetzt. Danach wandert das Element weiter nach oben in den Heap hinein, um schließlich die Heap-Eigenschaft wieder herzustellen, indem das Element ggfs. mit seinem Vater die Plätze tauscht. Auch diese Operation hat die Komplexität $\Theta(\log n)$.
Skizze:



Soll im Rahmen eines Greedy-Algorithmus zunächst ein Heap mit n Elementen hergestellt werden, so ist es günstiger, anstatt n-mal die Einfügeoperation durchzuführen, was die Komplexität $\log 1 + \log 2 + \cdots + \log n = \Omega(n \log n)$ hätte, die bei HeapSort beschriebene Operation zum Aufbau des Heaps durchzuführen (Phase 1 von HeapSort), welche nur die Komplexität $\mathcal{O}(n)$ hat.

Wir diskutieren nun die verschiedenen Greedy-Algorithmen hinsichtlich ihrer Komplexität, wenn ein solche effiziente priority queue-Implementierung verwendet wird.

Beim Bruchteil-Rucksackproblem wird zunächst ein Heap mit den v_i/g_i Werten aufgebaut (Komplexität $\mathcal{O}(n)$). In jedem Schleifendurchlauf wird das Maximum bestimmt und ausgelesen. Dies ergibt insgesamt die Komplexität $\mathcal{O}(n \log n)$ (was in diesem Fall auch nicht besser ist, als wenn man alle Elemente zunächst vollständig sortiert).

Analoge Komplexitätsabschätzungen gelten beim Auftragsplanungsproblem, also Komplexität $\mathcal{O}(n \log n)$.

Der Dijkstra-Algorithmus wurde schon ohne Verwenden von priority queues in der Komplexität mit $\mathcal{O}(|V|^2)$ abgeschätzt. Mittels Heap könnte man den Algorithmus wie folgt implementieren. Der Heap verwaltet die Menge W. Dieser wird zunächst mit den (vorläufigen) l-Werten der Knoten in V initialisiert (Komplexität $\mathcal{O}(|V|)$). Im Verlauf der $\mathcal{O}(|V|)$ vielen Schleifendurchläufe muss jedes Mal der Knoten mit minimalem l-Wert in W bestimmt und aus W entfernt werden (Komplexität $\mathcal{O}(\log |W|) = \mathcal{O}(\log |V|)$). (Der Heap ist dabei natürlich so organisiert, dass sich das ***Minimum*** an der Wurzel befindet).

Über alle Schleifendurchläufe hinweg müssen insgesamt $\mathcal{O}(|E|)$ viele l-Werte in W aktualisiert werden; dieses sind jeweils die Nachbarknoten v' von v. Um diese Knoten effizient aufzufinden, verwenden wir als Datenstruktur zur Repräsentation des Graphen G eine Adjazenzliste (vgl. Seite 81). Nachdem der l-Wert eines Knotens v' erniedrigt wurde, muss der Knoten dann im Heap evtl. weiter nach oben wandern – wie oben beschrieben (Komplexität $\mathcal{O}(\log|V|)$). Daher ergeben diese Heap-Modifikationen insgesamt einen Komplexitätsanteil von $\mathcal{O}(|E|\log|V|)$. Solcherart implementiert, hat der Dijkstra-Algorithmus daher die Komplexität $\mathcal{O}(|V|\log|V|) + \mathcal{O}(|E|\log|V|) = \mathcal{O}((|E| + |V|)\log|V|)$. Dies zahlt sich gegenüber $\mathcal{O}(|V|^2)$ dann aus, wenn der Graph relativ dünn besetzt ist, was seine Kantenzahl betrifft (also falls $|E| = \mathcal{O}(|V|^2/\log|V|)$).

Beim Kruskal-Algorithmus kann man ebenfalls vorteilhaft einen Heap verwenden, der zu Beginn mit den Kantenwerten gefüllt wird. In jedem Schritt wird dann diejenige Kante mit minimalem (bzw. maximalem) Wert bestimmt und aus dem Heap entfernt (Gesamtkomplexität für diese Heap-Operationen: $\mathcal{O}(|E|\log|E|)$). Nun tritt aber das Problem auf, dass getestet werden muss, ob diese Kante, zusammen mit den bereits ausgewählten Kanten einen Zyklus bildet. Um diesen Test effizient zu gestalten, benötigen wir eine geeignete Datenstruktur, die die sog. *Union-Find-Operationen* unterstützt. Hierbei soll eine Menge von Daten $v_1, \ldots, v_n$ verwaltet werden (in unserem Fall ist dies die Menge aller Knoten des Graphen). Jedes Objekt v_i wird zunächst im Initialzustand als eine einelementige Menge $\{v_i\}$ verstanden. Zu einem späteren Zeitpunkt können diese Mengen z.T. vereinigt worden sein, so dass im Allgemeinen also ein System von (disjunkten) Teilmengen von $\{v_1, \ldots, v_n\}$ verwaltet werden muss. Die Operationen *Union* und *Find* sind dann wie folgt definiert:

- *Find*(v_i): Es wird eine eindeutige Bezeichnung zurückgeliefert für die Menge, in der sich v_i befindet; dies kann zum Beispiel ein kanonisches Element dieser Menge sein.
- *Union*(v_i, v_j): Die beiden durch v_i und v_j identifizierten Mengen werden vereinigt und fortan als eine einzige Menge verwaltet.

Wenn man zum Beispiel testen will, ob sich die Elemente v_i und v_j in derselben Menge befinden, so kann man testen, ob *Find*(v_i) = *Find*(v_j) gilt. Im Falle des Kruskal-Algorithmus muss genau mit diesem Test festgestellt werden, ob eine Kante $\{v_i, v_j\}$ einen Zyklus mit bereits vorhandenen Kanten schließt (denn die disjunkten Teilmengen entsprechen genau den Zusammenhangskomponenten). Wenn dies nicht der Fall ist, und somit die Kante hinzugefügt werden kann, so muss diese Tatsache mittels der Operation *Union*(v_i, v_j) installiert werden.

Sei $f(n)$ die Komplexität von *Find* und sei $u(n)$ die Komplexität von *Union*, jeweils bezogen auf eine n-elementige Grundmenge. Dann lässt sich die Komplexität des Kruskal-Algorithmus durch $\mathcal{O}(|E|(\log|E| + f(|V|) + u(|V|)))$ abschätzen. Es gibt effiziente Union-Find-Datenstrukturen und dazugehörige Algorithmen (eine einfache Implementierung ist unten angegeben), so dass man von einer Komplexität von $\mathcal{O}(|E|\log|E|)$ für den Kruskal-Algorithmus ausgehen kann.

Sehr einfach lassen sich die Operationen *Union* und *Find* auf einer n-elementigen Grundmenge mittels eines Arrays $A[1..n]$ implementieren. Dieses Array wird so initia-

lisiert, dass $A[i] = i$ für alle i gilt. Dies bedeutet, dass von jedem Element i ein „Zeiger“ auf sich selbst ausgeht; das heißt, i ist gleichfalls das kanonische Element der Menge $\{i\}$. Im allgemeinen findet man das kanonische Element der Menge, in der sich Element i befindet, dadurch dass man $A[i]$, $A[A[i]]$, usw. solange sondiert, bis man ein k mit $k = A[k]$ vorfindet. Dann ist k das kanonische Element der betreffenden Menge. Das kanonische Element ist also immer die Baum-Wurzel (wenn man die im Array eingetragenen Werte als Zeiger betrachtet). Somit können wir *Union* und *Find* (einschließlich der Initialisierung) einfach implementieren:

Algorithmus 5.4 `UFinit()`

```
1: FOR i = 1 TO n DO
2:     A[i] = i
```

Algorithmus 5.5 `UFunion(`i, j`)`

```
1: z =random() ∈ [0, 1]
2: IF z = 0 THEN
3:     A[i] = j
4: ELSE
5:     A[j] = i
```

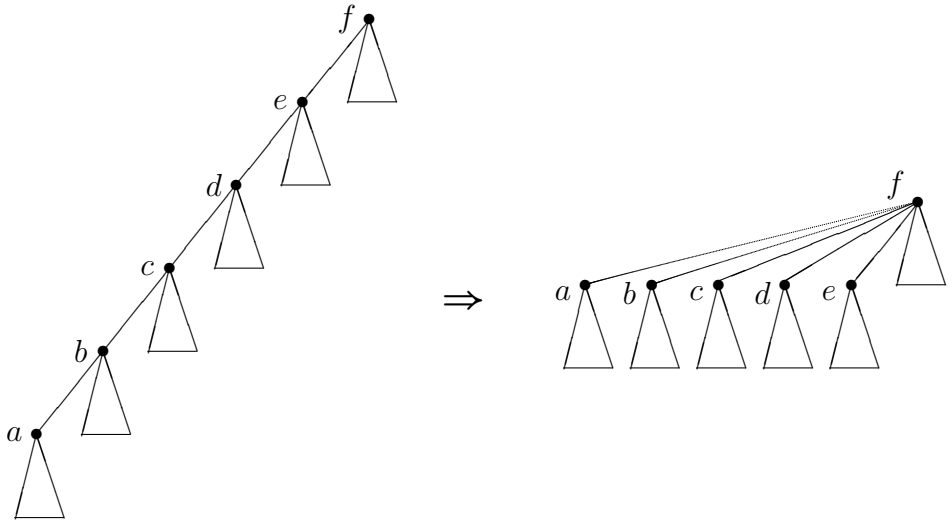
Algorithmus 5.6 `UFfind(`i`)`

```
1: IF i = A[i] THEN
2:     return i
3: ELSE
4:     j = UFfind(A[i])
5:     A[i] = j
6:     return j
```

Hier gibt es mehreres zu kommentieren: Bei der Union-Operation wird für die Korrektheit vorausgesetzt, dass sowohl i als auch j jeweils kanonische Elemente sind (das heißt, es muss $A[i] = i$ und $A[j] = j$ gelten). Ferner wird der Zufall verwendet, um festzulegen, ob die Menge i an die Menge j angekoppelt wird oder umgekehrt. Hierdurch soll verhindert werden, dass die Vereinigungsmengen in systematischer Weise unbalanciert werden. Bei einer Abfolge von mehreren Union-Operationen entstehen solcherart also „Zufallsbäume“.

Die Find-Operation ist im Prinzip so implementiert wie oben beschrieben: man sucht nach einem k mit $k = A[k]$. Als „Seiteneffekt“ wird hierbei die „Verzeigerungsstruktur“ umgebaut, so dass die Wege bis zur Wurzel sich verkürzen. In alle vorgefundenen $A[i]$-Werte wird ein direkter Zeiger auf die Baumwurzel eingetragen („Pfadkomprimierung“). Dies ist also ein Beispiel für eine *selbstorganisierende Datenstruktur*: je häufiger eine Find-Operation stattfindet, desto mehr sinkt dabei der Suchaufwand.

Skizze:



Anstelle bei der Union-Operation, wie vorgeschlagen, den Zufall zu Hilfe zu nehmen, kann man in einem Extra-Array speichern, wie viele Elemente sich hinter einem kanonischen Mengen-Element verbergen (also welche Mächtigkeit die betreffende Menge hat). Diese Information kann man dann bei der Union-Operation ausnützen, um die *kleinere* der beiden Mengen an die größere anzukoppeln, und nicht umgekehrt. Bei der Union-Operation muss diese Information beim kanonischen Element dann neu eingetragen werden (als Summe der vorherigen beiden Werte).

Für diese zuletzt vorgeschlagene Implementierung gilt, dass der nach n Union-Operationen entstehende Baum nicht tiefer sein kann als $\log_2(n+1)$. Dies lässt sich durch eine Induktion nach n einsehen. Der Induktionsanfang $n = 0$ ist klar: alle Bäume bestehen dann aus einem einzelnen Knoten, haben also Tiefe $\log_2 1 = 0$. Seien nun zwei Bäume gegeben mit jeweils n_1 bzw. n_2 Knoten, die vereinigt werden sollen, wobei $n_1 \leq n_2$ gilt, so dass die Wurzel des ersten Baumes als Sohn an die Wurzel des zweiten Baumes angehängt wird. Ferner hat nach Induktionsvoraussetzung der erste Baum die Tiefe $\leq \log_2 n_1$ und der zweite Baum die Tiefe $\leq \log_2 n_2$. Es sei $n = n_1 + n_2$ und es gilt also $n_1 \leq n/2$, also $\log_2 n_1 \leq \log_2 n - 1$. Dann ergibt diese Vereinigungsaktion eine Baumtiefe von $\leq \max\Big(1 + \log_2 n_1, \log_2 n_2\Big) \leq \log_2 n$. Daher können wir die worst-case Komplexität einer einzelnen Find-Operationen mit $\mathcal{O}(\log n)$ abschätzen, wobei der sich im amortisierten Sinne positiv auswirkende Effekt der Pfadkomprimierung noch nicht berücksichtigt wurde.

Tatsächlich lässt sich zeigen, dass eine beliebige Folge von $\mathcal{O}(n)$ Union- und Find-Operationen sich in der Komplexität mit $\mathcal{O}(n \log^* n)$ abschätzen lässt. Hierbei ist die Funktion $\log^* n$ wie folgt definiert:

$$\log^* n \;=\; \min\{k \mid \underbrace{\log \ldots \log}_{k\text{-mal}} n \leq 1\}$$

Die Funktion $\log^* n$ ist eine ungeheuer *langsam* anwachsende Funktion: für $n \leq 10^{19728}$ ist $\log^* n \leq 5$. Die Komplexität einer einzelnen *Find*-Operation kann also als (nahezu) konstante betrachtet werden.

Kapitel 6

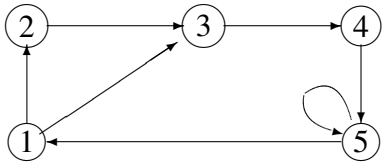
Algorithmen auf Graphen

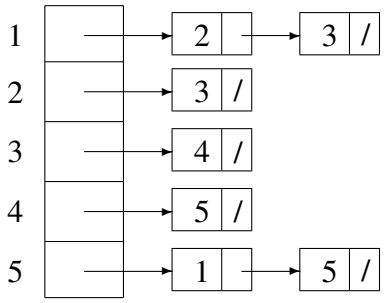
Es wurden im Zusammenhang mit Greedy-Algorithmen bereits einige Algorithmen auf Graphen besprochen. In diesem Kapitel sollen weitere Algorithmen betrachtet werden, die einige typische Aufgabenstellungen bei Graphen lösen, wie zum Beispiel das Durchsuchen eines Graphen, entweder auf die depth-first oder auf die breadth-first Weise, sowie das Bestimmen aller kürzesten Wege und das Bestimmen eines maximalen Flusses. Es wird in diesem Kapitel elementare Kenntnis der Graphentheorie vorausgesetzt.

6.1 Repräsentation von Graphen

Für die verschiedenen Algorithmen auf Graphen spielt es eine Rolle, in welcher Form der zu bearbeitende Graph repräsentiert wird. Die gewählte Form der Repräsentation kann sich auf die Komplexität des jeweiligen Algorithmus auswirken. Wir diskutieren hier zwei gängige Darstellungsformen, die *Adjazenzliste* und die *Adjazenzmatrix*. Die Adjazenzliste für einen Graphen $G = (V, E)$ besteht aus einem Array $A[1..|V|]$ von Zeigern. Diese Zeiger sind jeweils Ausgangspunkt für eine verkettete Liste. Die bei $A[i]$ startende Liste beschreibt die direkten Nachfolgerknoten von Knoten i.

Beispiel:
Der folgende gerichtete Graph wird durch die Adjazenzliste rechts repräsentiert.





Wir repräsentieren denselben Graphen als Nächstes durch seine Adjazenzmatrix. Dies ist eine Boolesche $|V| \times |V|$ Matrix und das Matrixelement an Position (i, j) enthält eine

1 genau dann, wenn eine Kante von i nach j vorhanden ist, also falls $(i, j) \in E$.

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

Der Speicheraufwand für die Adjazenzliste ist $\mathcal{O}(|V| + |E|)$, für die Adjazenzmatrix dagegen $\mathcal{O}(|V|^2)$. Die Adjazenzliste ist also in Bezug auf den Speicherplatz vorteilhaft, wenn der Graph *dünn besetzt* ist, also falls $|E| = o(|V|^2)$.

Ansonsten hängt die Wahl der Repräsentation von dem verwendeten Algorithmus ab, bzw. davon, welche Arten des Zugriffs auf den Graphen besonders effizient zu unterstützen sind. In den meisten Fällen ist die Adjazenzliste geeignet. Sollen allerdings für einen Knoten i seine *Vorgänger* aufgesucht werden, so ist diese Operation bei der Adjazenzliste nur ineffizient ausführbar.

Die Adjazenzliste ist vorteilhaft, wenn die Knoten bzw. Kanten in der natürlichen Reihenfolge, gegeben durch die Nachbarschaftsbeziehung auf dem Graphen, durchsucht werden sollen. Hier ist man bei der Matrizendarstellung und vor allem bei einem dünn besetzten Graph (wenige Kanten) im Nachteil, da man dann z.B. eine Zeile der Matrix solange durchsuchen muss, bis man eine 1 vorfindet (Aufwand $\mathcal{O}(n)$).

Die Adjazenzliste ist weiterhin von Vorteil, wenn man während des Algorithmenablaufs den Graph dynamisch verändern will, und Knoten oder Kanten, die schon „erfolgreich bearbeitet“ wurden, „ausklinken“ möchte, so dass für diese im weiteren Ablauf keine Komplexität mehr anfällt.

Steht dagegen die Abfrage „ist die Kante (x, y) vorhanden“ im Vordergrund, so ist diese Frage mittels der Adjazenzmatrix schneller zu beantworten als durch Adjazenzlisten, da die bei x startende Liste erst durchsucht werden muss.

Oft ist es vorteilhaft, den gegebenen Graphen in beiden Darstellungsformen zur Verfügung zu haben, oder eine der Darstellungsformen durch weitere Datenstrukturen (Suchbäume, Hashtabellen, Priority Queues, Union-Find-Strukturen) anzureichern. Beispiele hierfür haben wir beim Dijkstra-Algorithmus und Kruskal-Algorithmus schon kennengelernt.

Gelegentlich kommen bei manchen Anwendungen auch Graphen mit *Mehrfachkanten* und/oder Graphen mit *Schlingen* vor. Sollte dies bei den nachfolgenden Algorithmen der Fall sein, so weisen wir explizit darauf hin. Die obigen Datenstrukturen lassen sich entsprechend erweitern. Zum Beispiel können Mehrfachkanten genauso wie einfache Kanten mit ganzzahligen Gewichten behandelt werden.

In vielen Anwendungen sind den Kanten des Graphen gewisse Werte zugeordnet (zu interpretieren als Kosten, Längen, Kapazitäten, etc.) Diese Zusatzinformation lässt sich ohne weiteres in obige Datenstrukturen einbetten: Bei den Adjazenzlisten müssen zusätzlich zu den Knotennummern der Nachbarn die Kantenwerte untergebracht werden.

Bei den Adjazenzmatrizen gibt es zwei Möglichkeiten: Wenn die Kantenbewertung 0 nicht vorgesehen ist, also wenn zum Beispiel alle Kanten positive Werte haben, so kann

man mit dem Matrixeintrag 0 ausdrücken, dass die Kante nicht vorhanden ist. Und durch den Matrixeintrag $w > 0$ wird ausgedrückt, dass die betreffende Kante vorhanden ist, und dass ihr zugeordneter Wert = w ist.

Ansonsten könnte man die Adjazenzmatrix als ein Array $[1..|V|, 1..|V|]$ von Zeigern organisieren. Falls der Matrixeintrag = NIL ist, so ist die betreffende Kante nicht vorhanden. Wenn der Eintrag ein Zeiger $\neq$ NIL ist, so zeigt dieser auf den betreffenden Kantenwert.

Den Fall von *ungerichteten* Graphen kann man im Prinzip genauso (mittels Adjazenzlisten oder mittels Adjazenzmatrizen) behandeln, denn ein ungerichteter Graph kann als ein gerichteter Graph verstanden werden, in dem die Kante (u, v) genau dann vorhanden ist, wenn die Kante (v, u) vorhanden ist. Im Falle der Adjazenzmatrix wird die Matrix dann symmetrisch sein. (Man braucht in diesem Fall also evtl. nur die obere Dreiecksmatrix zu speichern).

Im Folgenden werden wir die Komplexitätsangaben bei Graphenalgorithmen immer in Abhängigkeit von der Knotenzahl $|V|$ und/oder der Kantenzahl $|E|$ machen.

6.2 Breiten- und Tiefensuche

Eine häufige Aufgabe ist, dass ein Graph systematisch durchsucht werden muss. Hierzu bieten sich zwei mögliche Strategien an, die *Breitensuche* (breadth-first search) und die *Tiefensuche* (depth-first search). Viele andere Graphalgorithmen benötigen einen solchen Suchalgorithmus als Unterprogramm; oder es ist so, dass in das Grundschema der Breiten- oder Tiefensuche weitere algorithmische Aktionen „eingeklinkt“ sind. Mit anderen Worten, viele andere Algorithmen auf Graphen können als Erweiterung der Breiten- oder Tiefensuche aufgefasst werden.

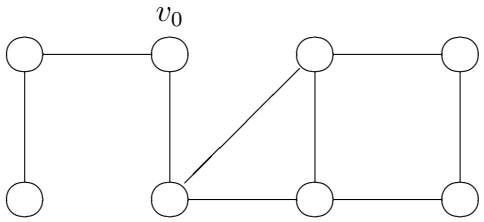
Die Breitensuche ist letztlich nichts anderes als der bereits besprochene Dijkstra-Algorithmus, wobei die Kantengewichte jetzt wegfallen, bzw. alle Kantenwerte auf 1 gesetzt sind. Wir formulieren den Algorithmus nochmals, wobei wir eine *(Warte-) Schlange* Q (first-in, first-out list) als Datenstruktur verwenden. Einer der Knoten $v_0 \in V$ ist der Startknoten, von dem aus die Suche beginnt. Mit $Adj(u)$ bezeichnen wir die direkten Nachbarn von Knoten u. Die Tabelle d enthält die kürzesten Abstände von v_0 aus.

Algorithmus 6.1 `breitensuche`(v_0)

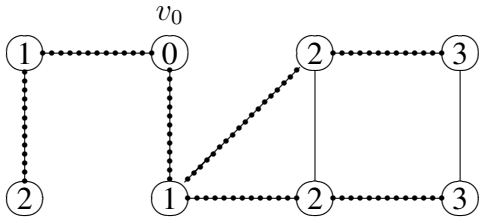
```
 1: FOR v ∈ V DO
 2:    d[v] = ∞
 3: d[v_0] = 0
 4: Q = {v_0}
 5: WHILE Q ≠ ∅ DO
 6:    u =erstes Elementn in Q
 7:    Entferne u aus Q
 8:    FOR v ∈ Adj(u) DO
 9:       IF d[v] = ∞ THEN
10:          d[v] = d[u] + 1
11:          Q = Q ∪ {v}
```

Ausgehend vom Knoten v_0 werden zunächst alle seine Nachbarn der Reihe nach betrachtet (dann die Nachbarn der Nachbarn, usw.) und es wird ihr minimaler Abstand zu v_0 in der Tabelle d eingetragen. Implizit wird dabei ein Baum mit Wurzel v_0 konstruiert; dieser enthält immer dann die Kante (u, v), wenn der THEN-Zweig betreten wird.

Beispiel:
Gegeben sei folgender Graph, in dem der Startknoten v_0 besonders gekennzeichnet ist.



Nach Ablauf der Breitensuche ergeben sich folgende (in den Knoten eingetragene) d-Werte. Ferner ist der Baum, der die Wege minimaler Länge von v_0 aus charakterisiert, gestrichelt gezeichnet.



Da jeder Knoten und jede Kante einmal besucht wird, ist die Komplexität der Breitensuche $\mathcal{O}(|V| + |E|)$. Falls der Graph zusammenhängend ist, so gilt $|E| \geq |V| - 1$. In diesem Fall können wir die Komplexität somit durch $\mathcal{O}(|E|)$ abschätzen.

Bei der Tiefensuche wird „in die Tiefe“ zuerst gegangen. Am einfachsten ist dies durch eine rekursive Prozedur zu formulieren, die vom Hauptprogramm aus mit dem Startkno-

ten v_0 aufzurufen ist. Jeder Knoten erhält eine „Farbe“, wobei *weiß* bedeutet, dass der Knoten noch nicht besucht wurde; *grau* bedeutet, dass der Knoten besucht wurde, aber noch nicht endgültig abgeschlossen ist; wenn die Farbe eines Knotens *schwarz* ist, so wurde dieser endgültig bearbeitet. Zu Beginn seien alle Knoten weiß.

Algorithmus 6.2 `tiefensuche`(u)

```
1: farbe[u]=grau
2: FOR v ∈ Adj(u) DO
3:    IF farbe[v]=weiß THEN
4:       tiefensuche(v)
5: farbe[u]=schwarz
```

Die Komplexität der Tiefensuche kann wie bei der Breitensuche mit $\mathcal{O}(|E|)$ abgeschätzt werden.

6.3 Topologisches Sortieren

Eine *topologische Sortierung* eines *gerichteten* Graphen $G = (V, E)$ ist eine Anordnung seiner Knoten $V = \{v_1, v_2, \ldots, v_n\}$, so dass für alle Kanten (v_i, v_j) gilt: $i < j$.
Graphen, die Zyklen enthalten, können offensichtlich nicht topologisch sortiert werden. Kann man wenigstens alle azyklischen Graphen topologisch sortieren?

SATZ:
Ein Graph G ist azyklisch genau dann, wenn er sich topologisch sortieren lässt.

Beweis:

($\Leftarrow$) Klar.

($\Rightarrow$) Sei G ein azyklischer Graph. Dann muss G einen Startknoten v, also einen Knoten ohne Vorgänger besitzen. (Wenn nicht: konstruiere einen Kreis, indem man von Vorgänger zu Vorgänger läuft). Wir geben dem Knoten v die Nummer 1, also $v = v_1$. Nun entfernen wir v (samt seiner hinausgehenden Kanten) aus G und sortieren den Restgraphen topologisch, beginnend mit v_2. □

In dem Beweis ist auch ein Algorithmus enthalten zur Konstruktion einer topologischen Sortierung: Bestimme einen Startknoten, entferne diesen, bestimme wieder einen Startknoten, usw.
Es geht aber noch eleganter mit Hilfe der Tiefensuche. Wir betten den Tiefensuch-Algorithmus in folgendes Programm `search`(V) ein:

Algorithmus 6.3 `search`(V)

```
1: FOR v ∈ V DO
2:    farbe[v]=weiß
3: FOR v ∈ V DO
4:    IF farbe[v]=weiß THEN
5:       tiefensuche(v)
```

Seien nun die Knoten anhand der umgekehrten Reihenfolge des Schwarz-werdens (was vor Verlassen der Prozedur Tiefensuche passiert) durchnummeriert:

$$V = \{v_1, v_2, \ldots, v_n\}$$

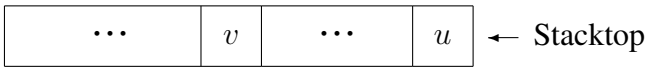
(Das heißt, v_n wird als erster schwarz und v_1 als letzter). Die Behauptung ist: dies liefert (bei einem azyklischen Graph als Eingabe) eine topologische Sortierung des Graphen.

Dies kann man wie folgt einsehen. Betrachte eine beliebige Kante (u, v) des Graphen. Die Nummerierung der Knoten u und v sollte am Ende solcherart sein, dass die Nummer von u kleiner als die von v ist. In anderen Worten, der Knoten v sollte bei einem Aufruf von `search`(V) *vor* dem Knoten u schwarz werden. Im Verlauf von `search`(V) wird irgendwann `tiefensuche`(u) aufgerufen. Hierbei wechselt u die Farbe von weiß nach grau. Wir betrachten nun den Zeitpunkt, bei dem im Rahmen der FOR-Schleife der Knoten $v \in Adj(u)$ in Betracht gezogen wird. Welche Farbe hat v zu diesem Zeitpunkt?

Fall 1: v ist schwarz. Dann ist v offensichtlich bereits *vor* u schwarz geworden, denn u wird erst am Ende von `tiefensuche`(u) schwarz.

Fall 2: v ist grau. Das bedeutet, dass bereits zuvor ein Aufruf von `tiefensuche`(u) stattgefunden hat, und dieser Aufruf noch nicht abgeschlossen ist, da v noch nicht schwarz ist. Das heißt, diese Inkarnation von Tiefensuche liegt sozusagen „tiefer“ im Stack der rekursiven Aufrufe als der aktuelle Aufruf `tiefensuche`(u).

Skizze:



Das bedeutet, dass es einen Pfad von v nach u geben muss. Zusammen mit der Kante (u, v) ergibt dies einen Kreis. Ein solcher kann in einem azyklischen Graphen nicht auftreten. Daher kann dieser Fall gar nicht eintreten.

An dieser Stelle wollen wir die Bemerkung einschieben, dass man das Vorfinden eines grauen Knoten in der FOR-Schleife von Tiefensuche als Indikator für einen vorhandenen Zyklus verwenden kann. Daher lässt sich aus dieser Beobachtung auch leicht ein Algorithmus zum Testen, ob ein Graph azyklisch ist oder nicht, ableiten.

Fall 3: v ist weiß. Dann erfolgt ein rekursiver Aufruf von `tiefensuche`(v). Erst wenn dieser ausgeführt ist, was den Effekt hat, dass v schwarz geworden ist, kehrt die Kontrolle in die betrachtete Inkarnation von `tiefensuche`(u) zurück. Daher wird auch in diesem Fall der Knoten v *vor* dem Knoten u schwarz.

Die Komplexität von `search`(V) ist $\mathcal{O}(|V|+|E|)$, da im Hauptprogramm jeder Knoten und im Innern von Tiefensuche jede Kante genau einmal „angefasst“ wird.

6.4 Transitive Hülle: Warshall-Algorithmus

Die *transitive Hülle* eines Graphen $G = (V, E)$ ist definiert als der Graph $G^* = (V, E^*)$ mit

$$(u, v) \in E^* \Leftrightarrow \text{ es gibt einen Pfad in } G \text{ von } u \text{ nach } v$$

Wir können den Graph G^* im Prinzip mit den bekannten Algorithmen berechnen, indem man z.B. von jedem Startknoten $u \in V$ aus eine Breitensuche durchführt. Es gibt aber einen wesentlich einfacheren und eleganteren Algorithmus für diese Aufgabe, der von Warshall stammt.

Sei der Graph $G = (V, E)$ mit $V = \{1, 2, \ldots, n\}$ gegeben. Betrachten wir die Booleschen Variablen $m_{i,j}^k$, die wie folgt definiert sind:

$$m_{i,j}^k = \begin{cases} 1, & \text{es gibt einen Pfad von } i \text{ nach } j, \text{ wobei} \\ & \text{alle zwischen } i \text{ und } j \text{ liegenden Knoten} \\ & \text{aus der Menge } \{1, \ldots, k\} \text{ stammen} \\ 0, & \text{sonst} \end{cases}$$

Es ist $m_{i,j}^0 = 1$ genau dann, wenn $(i, j) \in E$ (oder wenn $i = j$). Ferner gilt folgende rekursive Beziehung

$$m_{i,j}^k = m_{i,j}^{k-1} \vee (m_{i,k}^{k-1} \wedge m_{k,j}^{k-1})$$

die man sich anhand der Definition von $m_{i,j}^k$ leicht klarmacht.

Es gilt $(i, j) \in E^*$ genau dann, wenn $m_{i,j}^n = 1$. Daher kann man die transitive Hülle berechnen, indem man die Matrizen $(m_{i,j}^k)$ für $k = 0, 1, \ldots, n$ berechnet. Hierbei entspricht $(m_{i,j}^0)$ gerade der Ausgangskantenmenge E. (Der Algorithmus fällt in die Kategorie „dynamisches Programmieren“, vgl. Kapitel 4).

Der folgende Algorithmus von Warshall führt diese Idee durch. Sei $A[1..n, 1..n]$ die Adjazenzmatrix des Graphen G. Nach Ausführung des Algorithmus stellt A die Adjazenzmatrix von G^* dar.

Algorithmus 6.4 `transitiveHuelle`(A)

```
1: FOR k = 1 TO n DO
2:   FOR i = 1 TO n DO
3:     IF A[i,k] THEN
4:       FOR j = 1 TO n DO
5:         IF A[k,j] THEN
6:           A[i,j] =TRUE
```

Offensichtlich ist die Komplexität dieses Algorithmus $\Theta(n^3)$.

Der Warshall-Algorithmus kann geringfügig modifiziert werden, um in einem kantenbewerteten Graphen (wobei ein Kantenwert die „Länge“ der Kante repräsentieren soll), die kürzesten Wege (und deren Längen) zwischen *allen* Knotenpaaren (i, j) zu berechnen.

Die Aufgabe unterscheidet sich insofern vom Dijkstra-Algorithmus, da dieser von *einem* Startknoten aus alle kürzesten Wege zu den anderen Knoten berechnet.

Es bedeute $m_{i,j}^k$ jetzt die kürzeste Weglänge von i nach j, wobei als mögliche „Zwischenknoten“ nur die Knoten in $\{1, , \ldots, k\}$ zugelassen sind. Ähnlich wie oben gilt die rekursive Beziehung

$$m_{i,j}^k \;=\; \min\left(m_{i,j}^{k-1}\,,\, m_{i,k}^{k-1} + m_{k,j}^{k-1}\right)$$

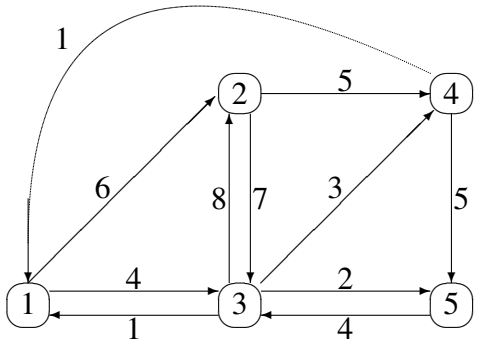
Diese Rekursion kann völlig analog zum obigen Warshall-Algorithmus in einen Algorithmus umgesetzt werden (dieser stammt von Floyd). Sei nun $E[1..n, 1..n]$ eine Matrix, die die Kanten-Längen angibt; und wenn zwischen i und j keine Kante besteht, so sei $E[i,j] = \infty$. Ferner sei $E[i,i] = 0$. Nach Ausführung des Algorithmus enthält $E[i,j]$ die kürzeste Weglänge von i nach j.

Algorithmus 6.5 `floydWarshall`(A)

```
1: FOR k = 1 TO n DO
2:   FOR i = 1 TO n DO
3:     FOR j = 1 TO n DO
4:       IF E[i,k] + E[k,j] < E[i,j] THEN
5:         E[i,j] := E[i,k] + E[k,j]
```

Die kürzesten Wege selbst erhält man, indem man darüber Buch führt, wie die E-Werte zu Stande kamen.

Beispiel:
Gegeben sei folgender Graph:



Die folgenden Matrizen stellen „Momentaufnahmen“ im Verlauf des Algorithmus dar. Vor jedem Erhöhen des k-Werts wird eine solche Momentaufnahme gemacht:

$$k = 0: \begin{bmatrix} 0 & 6 & 4 & \infty & \infty \\ \infty & 0 & 7 & 5 & \infty \\ 1 & 8 & 0 & 3 & 2 \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & 4 & \infty & 0 \end{bmatrix} \quad k = 1: \begin{bmatrix} 0 & 6 & 4 & \infty & \infty \\ \infty & 0 & 7 & 5 & \infty \\ 1 & 7 & 0 & 3 & 2 \\ 1 & 7 & 5 & 0 & 5 \\ \infty & \infty & 4 & \infty & 0 \end{bmatrix}$$

$$k=2: \begin{bmatrix} 0 & 6 & 4 & 11 & \infty \\ \infty & 0 & 7 & 5 & \infty \\ 1 & 7 & 0 & 3 & 2 \\ 1 & 7 & 5 & 0 & 5 \\ \infty & \infty & 4 & \infty & 0 \end{bmatrix} \quad k=3: \begin{bmatrix} 0 & 6 & 4 & 7 & 6 \\ 8 & 0 & 7 & 5 & 9 \\ 1 & 7 & 0 & 3 & 2 \\ 1 & 7 & 5 & 0 & 5 \\ 5 & 11 & 4 & 7 & 0 \end{bmatrix}$$

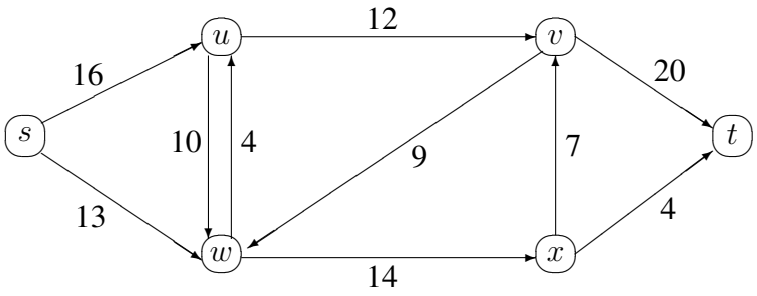
$$k=4: \begin{bmatrix} 0 & 6 & 4 & 7 & 6 \\ 6 & 0 & 7 & 5 & 9 \\ 1 & 7 & 0 & 3 & 2 \\ 1 & 7 & 5 & 0 & 5 \\ 5 & 11 & 4 & 7 & 0 \end{bmatrix} \quad k=5: \begin{bmatrix} 0 & 6 & 4 & 7 & 6 \\ 6 & 0 & 7 & 5 & 9 \\ 1 & 7 & 0 & 3 & 2 \\ 1 & 7 & 5 & 0 & 5 \\ 5 & 11 & 4 & 7 & 0 \end{bmatrix}$$

Beispielsweise sieht man, dass zunächst keine Kante von Knoten 2 nach 1 existiert, daher der Wert ∞. Später ergibt sich als kürzeste gefundene Verbindung die Länge 8 (=7+1), welche im weiteren Verlauf des Verfahrens noch auf 6 (=5+1) reduziert wird.

6.5 Flüsse in Netzwerken: Ford-Fulkerson

Wir wollen folgende Aufgabe lösen: Gegeben sei ein gerichteter Graph, in dem zwei Knoten besonders ausgezeichnet sind. Der eine Knoten s heißt die *Quelle* und ein anderer Knoten t heißt die *Senke*. Ferner sind den Kanten des Graphen positive, ganze Zahlen zugeordnet. Diese heißen *Kapazitäten*. Wir können den Graphen interpretieren als ein Netzwerk von Röhren, Straßen oder Leitungen, durch die elektrischer Strom, Fahrzeuge oder eine Flüssigkeit fließen kann. Die Kapazitäten entsprechen dann einer maximalen Durchlaufkapazität (z.B. in Liter pro Sekunde) oder einem maximalen Strom (z.B. in Ampère), den die betreffende Leitung verkraftet. Aus der Quelle strömt nun Flüssigkeit in die Röhren (bzw. Strom in die Leitungen, usw.) Diese durchläuft die Kanten und endet schließlich in der Senke.

Beispiel:



Die Aufgabe besteht nun darin, den Kanten konkrete Flusswerte (Ströme in Ampère, etc.) zuzuordnen, die die jeweiligen Kantenkapazitäten nicht überschreiten, so dass der Gesamtfluss von der Quelle zur Senke maximiert wird. Hierbei ist natürlich die Nebenbedingung einzuhalten, dass aus einem Knoten nicht mehr (oder weniger) herausfließen kann als hineinfließt.

Die folgende Definition führt diese intuitiven Vorstellungen präzise aus.

DEFINITION ((Fluss-)Netzwerk):
Ein (Fluss-) Netzwerk *ist ein gerichteter Graph* $G = (V, E)$ *mit zwei ausgezeichneten Knoten* $s, t \in V$, *der* Quelle *und der* Senke, *sowie einer Funktion* $c : V \times V \to \mathbb{N}$, *wobei für* $(u, v) \notin E$ *gilt* $c(u, v) = 0$. *Der Wert* $c(u, v)$ *repräsentiert die* Kapazität *der Kante* (u, v).
Ein (zulässiger) Fluss *für ein solches Netzwerk ist eine Funktion* $f : V \times V \to \mathbb{Z}$, *die folgende Bedingungen erfüllt:*

Kapazitätsbedingung $f(u, v) \leq c(u, v)$

Symmetriebedingung $f(u, v) = -f(v, u)$

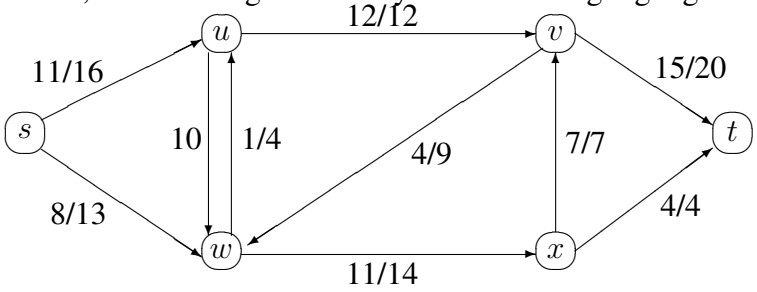
Kirchhoffsches Gesetz $\forall u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0$

Der Betrag *eines Flusses* f *ist* $|f| := \sum_{v \in V} f(s, v)$.

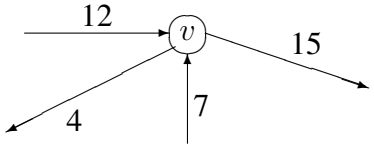
Die einem Netzwerk mit Kapazitätsfunktion zugeordnete Optimierungsaufgabe besteht darin, einen zulässigen Fluss f *zu finden, so dass dessen Betrag* $|f|$ *maximal ist.*

Aus der Definition ergibt sich, wenn zwischen zwei Knoten u und v weder die Kante (u, v) noch (v, u) vorhanden ist, so gilt $c(u, v) = c(v, u) = f(u, v) = f(v, u) = 0$.

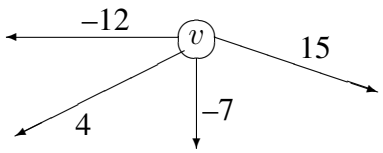
Beispiel:
In dem folgenden Diagramm ist ein zulässiger Fluss für obiges Beispiel eingetragen. Bei den Zahlenangaben ist die erste Zahl der Fluss und die zweite Zahl die Kapazität. Ist kein Fluss-Wert angegeben, so ist dieser gleich 0. Außerdem nicht eingetragen sind die negativen Flüsse, die sich aufgrund der Symmetriebedingung ergeben.



Dieser Fluss hat den Betrag $11 + 8 = 19$. Dies ist allerdings nicht der maximal-mögliche Fluss, wie wir gleich sehen werden.
Prüfen wir nach, dass das Kirchhoffsche Gesetz erfüllt ist. Nehmen wir als Beispiel den Knoten v:

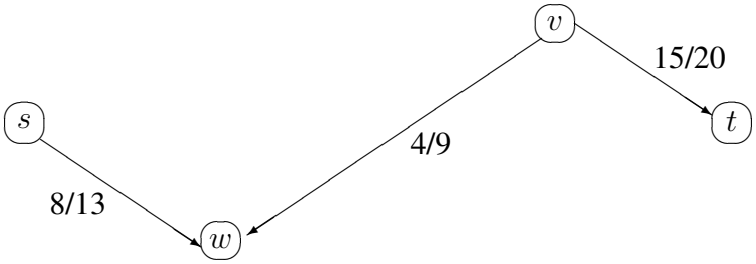


In den Knoten v geht insgesamt ein Fluss von $12 + 7 = 19$ hinein und es fließen entsprechend $4 + 15 = 19$ wieder heraus. Das Kirchhoffsche Gesetz ist also erfüllt. Im formalen Sinne der Kirchhoffbedingung müssten alle aus v *hinausgehenden* Flüsse aufsummiert werden:

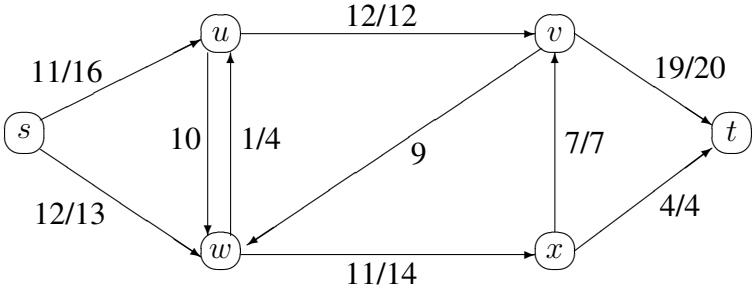


Die Summe dieser Flüsse ist tatsächlich gleich Null.

Der beim obigen Beispielgraphen eingetragene Fluss ist noch nicht optimal, denn betrachten wir mal die folgende Pfad-Verbindung zwischen s und t:



Entlang dieses Pfades kann der Fluss vergrößert werden; und zwar auf der ersten Kante um $13 - 8 = 5$ und ebenso auf der letzten Kante um $20 - 15 = 5$. Die mittlere Kante zeigt in die „falsche“ Richtung. Auch hier kann der Fluss in die „Vorwärtsrichtung“ vergrößert werden, indem man den (vorwärts gerichteten) negativen Fluss mit dem Wert –4 wieder bis auf 0 reduziert. Also ist auf der mittleren Kante (in diesem Sinne) eine Flussverbesserung um 4 möglich. Indem wir den Fluss nun um $\min(5, 4, 5) = 4$ auf allen Kanten dieses Pfades erhöhen – auf der mittleren Kante, da sie entgegengerichtet ist, um 4 Werte erniedrigen – erhalten wir den folgenden Graphen, der den Fluss-Betrag $11 + 12 = 23$ hat.

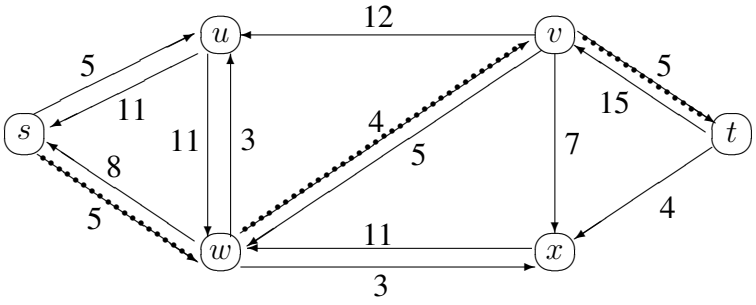


Tatsächlich ist der jetzt eingetragene Fluss maximal.

Was wir gerade eben durchgeführt haben, lässt sich systematischer mit dem Konzept des *Restnetzwerks* formulieren.

DEFINITION (Restnetzwerk):
Gegeben sei ein Netzwerk G mit Kapazität c und ein zulässiger Fluss f. Das zugeordnete Restnetzwerk G_f *mit einer zugeordneten* Restkapazität $c_f(u,v) = c(u,v) - f(u,v)$ *ist gegeben durch $G_f = (V, E_f)$, wobei $E_f = \{(u,v) \mid c_f(u,v) > 0\}$.*

Beispiel:
Das dem obigen Beispielgraphen auf Seite 90 zugeordnete Restnetzwerk ist das folgende:



Man erkennt, dass im Restnetzwerk auch Kanten (u,v) entstehen können, die zuvor im eigentlichen Netzwerk nicht vorhanden waren. (Dies ist nur dann der Fall, wenn $(v,u) \in E$).

Im obigen Diagramm ist gestrichelt derjenige Pfad eingezeichnet, entlang dessen wir zuvor den Fluss erhöht hatten. Hierzu die folgende Definition.

DEFINITION (Erweiterungspfad, Restkapazität):
Ein Erweiterungspfad *ist ein einfacher (also zyklenfreier) Pfad p in G_f von s nach t. Nach Definition von G_f muss für alle Kanten (u,v) auf p gelten $c_f(u,v) > 0$. Die* Restkapazität *von p ist $c_f(p) = \min\{c_f(u,v) \mid (u,v)$ liegt auf $p\}$.*

Wir bemerken, dass

$$g(u,v) = \begin{cases} c_f(p), & (u,v) \text{ liegt auf } p \\ 0, & \text{sonst} \end{cases}$$

ein zulässiger Fluss in G_f ist.

Lemma:
Sei G ein Netzwerk, c eine Kapazität, und f ein zulässiger Fluss. Sei G_f das zugehörige Restnetzwerk und c_f die zugehörige Restkapazität. Sei ferner g ein zulässiger Fluss auf G_f.
Dann gilt: $(f+g)$ ist ein zulässiger Fluss auf G mit $|(f+g)| = |f| + |g|$.

Beweis:
Wir überprüfen, dass $(f+g)$ ein zulässiger Fluss auf G ist.
Kapazitätsbedingung: Da g zulässiger Fluss in G_f ist, gilt $g(u,v) \leq c_f(u,v) = c(u,v) - f(u,v)$. Daraus folgt $(f+g)(u,v) = f(u,v) + g(u,v) \leq c(u,v)$.

Symmetriebedingung: $(f+g)(u,v) = f(u,v)+g(u,v) = -f(v,u)-g(v,u) = -(f(v,u)+g(v,u)) = -(f+g)(v,u)$.
Kirchhoffsches Gesetz: Sei $u \in V - \{s,t\}$. Dann gilt $\sum_{v\in V}(f+g)(u,v) = \sum_{v\in V}(f(u,v)+g(u,v)) = \sum_{v\in V} f(u,v) + \sum_{v\in V} g(u,v) = 0+0 = 0$.
Ferner gilt: $|(f+g)| = \sum_{v\in V}(f+g)(s,v) = \sum_{v\in V} f(s,v) + \sum_{v\in V} g(s,v) = |f| + |g|$.
Damit ist das Lemma bewiesen. □

Aufgrund des Lemmas können wir also einen bestehenden Fluss f in dem Netzwerk G mit Kapazität c dadurch verbessern, dass wir zunächst das Restnetzwerk G_f bestimmen; danach suchen wir nach einem Erweiterungspfad (also einem einfachen Pfad p in G_f von s nach t), bestimmen $c_f(p)$ und verbessern dann den Fluss f, indem wir den Fluss

$$g(u,v) = \begin{cases} c_f(p), & (u,v) \text{ liegt auf } p \\ 0, & \text{sonst} \end{cases}$$

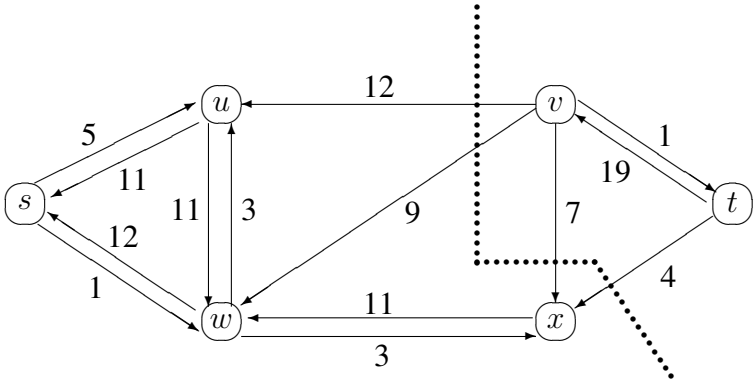
hinzuaddieren.
Genau dieses ist die Strategie des *Algorithmus von Ford-Fulkerson:*

Algorithmus 6.6 `fordFulkerson`(V,s,t)

1: Setze $f(u,v) = 0$ für alle $(u,v) \in V \times V$
2: **REPEAT**
3: Berechne G_f
4: Bestimme einen Erweiterungspfad p in G_f
5: $f = f + g$ *//Definition von g siehe oben*
6: **UNTIL** es gibt keinen Pfad in G_f von s nach t
7: gib f als maximalen Fluss aus

Durch das Lemma wird gerade gezeigt: Wenn ein Erweiterungspfad gefunden werden kann, so bringt der neue Fluss $(f+g)$ eine echte Verbesserung. Die zu klärende Frage ist allerdings noch: Wenn der bisherige Fluss f nicht maximal ist, kann dann immer ein Erweiterungspfad gefunden werden? Die (positive) Antwort wird durch das Min-Cut-Max-Flow-Theorem weiter unten gegeben.

Beispiel:
Nach Anwendung der Fluss-Erweiterung auf das Netzwerk auf Seite 90 erhalten wir das Netzwerk auf Seite 91. Nun bilden wir wieder das zugehörige Netzwerk G_f. Man erkennt, dass nun kein Pfad mehr von s nach t führt.



Die gestrichelte Linie stellt einen *Schnitt* durch das Netzwerk dar. Nur die Knoten links dieses Schnitts sind von der Quelle s aus erreichbar.

Der Begriff des Schnitts erweist sich auch weiterhin als nützlich.

DEFINITION (Schnitt, Kapazität):
Ein Schnitt *ist eine Zerlegung der Knotenmenge* $V = A \cup B$, $A \cap B = \emptyset$, *mit* $s \in A$ *und* $t \in B$.

Die Kapazität *eines Schnitts* (A, B) *ist* $c(A, B) = \sum_{u \in A, v \in B} c(u, v)$.

Sei f *ein Fluss. Dann ist der* Fluss über den Schnitt (A, B) *definiert durch* $f(A, B) = \sum_{u \in A, v \in B} f(u, v)$.

Es ist klar, dass für jeden zulässigen Fluss f gilt $f(A, B) \leq c(A, B)$, denn für jeden der Summanden gilt $f(u, v) \leq c(u, v)$.

Ferner gilt nach Definition des Fluss-Betrags: $|f| = f(\{s\}, V - \{s\})$. Tatsächlich spielt es für den Wert von $f(A, B)$ keine Rolle, an welcher Stelle wir das Netzwerk auseinanderschneiden. Für jeden Schnitt (A, B) gilt $|f| = f(A, B)$. Dies lässt sich leicht durch Induktion nach $|A|$ beweisen. Der Fall $|A| = 1$ ist gerade die Definition von $|f|$. Sei nun (A, B) ein Schnitt mit $|A| = n + 1$. Wir wählen einen Knoten $v \in A - \{s\}$ und betrachten den Schnitt $(A', B') = (A - \{v\}, B \cup \{v\})$. Nach Induktionsvoraussetzung gilt $f(A', B') = |f|$. Ferner gilt

$$f(A, B) = f(A', B') - \sum_{u \in A} f(u, v) + \sum_{u \in B} f(v, u) = |f| + \sum_{u \in V} f(v, u) = |f|$$

Somit gilt für jeden Schnitt (A, B) und jeden zulässigen Fluss f: $|f| \leq c(A, B)$. Anders ausgedrückt:

$$\max_f |f| \leq \min_{(A,B)} c(A, B)$$

Der folgende Satz besagt, dass es einen Fluss f^* gibt, bei dem obige Ungleichung zur Gleichung wird, also $|f^*| = \min_{(A,B)} c(A, B)$.

SATZ (Min-Cut-Max-Flow-Theorem):
Sei f *ein zulässiger Fluss in dem Netzwerk* G *mit Kapazität* c. *Dann sind die folgenden Aussagen äquivalent:*

(1) f *ist ein maximaler Fluss in* G*,*

(2) Das Restnetzwerk G_f *enthält keinen Erweiterungspfad,*

(3) Es gilt $|f| = c(A, B)$ *für einen Schnitt* (A, B) *von* G*.*

Beweis:

$(1) \Rightarrow (2)$ Dies folgt aus obigem Lemma: Wenn es in G_f einen Erweiterungspfad gibt, dann kann f echt verbessert werden, kann also nicht maximal gewesen sein.

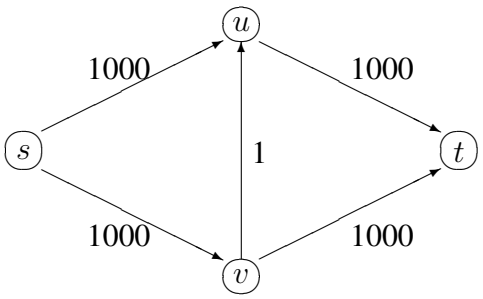
$(2) \Rightarrow (3)$ Setze $A = \{u \in V \mid$ es gibt einen Pfad in G_f von s nach $u\}$ und $B = V - A$. Dann gilt $s \in A$ und nach Voraussetzung $t \in B$. Also ist (A, B) ein Schnitt. Für alle Knoten $u \in A$ und $v \in B$ gilt: $0 = c_f(u, v) = c(u, v) - f(u, v)$, also $c(u, v) = f(u, v)$. Damit folgt:

$$|f| = f(A, B) = \sum_{u \in A, v \in B} f(u, v) = \sum_{u \in A, v \in B} c(u, v) = c(A, B)$$

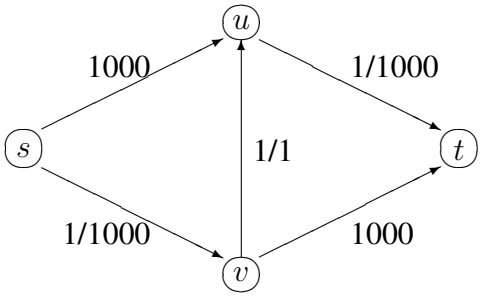
$(3) \Rightarrow (1)$ Für alle zulässigen Flüsse f und Schnitte (A, B) gilt $|f| \leq c(A, B)$. Falls also $|f| = c(A, B)$ gilt, so muss f ein maximaler Fluss sein. □

Durch diesen Satz wird die Korrektheit des Algorithmus von Ford-Fulkerson gezeigt: Es verblieb zu zeigen, dass es für jedes nicht-maximale f einen Erweiterungspfad in G_f gibt. Dies ist gerade $\neg(1) \Rightarrow \neg(2)$, oder gleichwertig: $(2) \Rightarrow (1)$.

Die Formulierung des Ford-Fulkerson Algorithmus ist an einer Stelle unterspezifiziert; es wird nämlich nicht gesagt, *welcher* Erweiterungspfad gewählt werden soll, sofern es mehrere gibt. Durch „ungeschickte“ Wahl des Erweiterungspfades kann Ford-Fulkerson eine recht lange Laufzeit haben, wie das folgende Beispiel zeigt.



Eine ungeschickte Wahl des Erweiterungspfades wäre s-v-u-t. Dann gilt $c_f(p) = 1$. Das heißt, nach einer Ausführung der REPEAT-Schleife sieht der Fluss wie folgt aus:



Im nächsten Schritt könnte der Pfad s-u-v-t gewählt werden, usw. Solcherart kommt der Algorithmus erst nach 2000 Schleifendurchläufen zum Stillstand, obwohl es in diesem Fall auch mit 2 Schleifendurchläufen gegangen wäre (nämlich s-u-t und s-v-t). Man kann den worst-case des Ford-Fulkerson Algorithmus somit lediglich mit $\mathcal{O}(|f^*| \cdot |E|)$ abschätzen, wobei f^* der maximale Fluss ist, denn im schlechtesten Fall benötigt der Algorithmus $|f^*|$ Schleifendurchläufe. Pro Schleifendurchlauf fällt der Aufwand für eine Suche nach einem Erweiterungspfad an. Dies ist $\mathcal{O}(|E|)$, wenn man dies mittels Breitensuche oder Tiefensuche realisiert.

Ungünstig ist, dass der Betrag des maximalen Flusses mit in die Komplexität einfließt. (Im Sinne der Bit-Komplexität kann dies sogar exponentielle Komplexität bedeuten). In dem Spezialfall allerdings, dass die in Frage kommenden Flüsse an den Kanten nur die Werte 0 oder 1 annehmen können (weil die Kapazitätswerte ≤ 1 sind), folgt $|f^*| \leq |E|$, so dass sich die Komplexität des Verfahrens dann mit $\mathcal{O}(|E|^2)$ abschätzen lässt. (Tatsächlich lässt sich der Algorithmus samt der dazugehörigen Abschätzung in diesem Spezialfall noch auf $\mathcal{O}(|V|^{2/3}|E|)$ verbessern). Dieser Spezialfall liegt zum Beispiel beim Matching-Problem vor, das im nächsten Abschnitt behandelt wird.

Die Abhängigkeit der Komplexität des Verfahrens vom Wert des maximalen Flusses kann allerdings vermieden werden. Die *Edmonds-Karp-Strategie* besteht darin, als Erweiterungspfad immer den *kürzestmöglichen* auszuwählen, also den mit der geringsten Kantenzahl. Ein solcher kürzester Pfad von s nach t in G_f wird zum Beispiel mit Hilfe der Breitensuche (=Dijkstra-Algorithmus) gefunden. Edmonds und Karp argumentieren, dass bei dieser Strategie nie mehr als $|V| \cdot |E|$ viele Schleifendurchläufe notwendig werden, so dass die Gesamtkomplexität dann auf $|V| \cdot |E| \cdot \mathcal{O}(|E|) = \mathcal{O}(|V| \cdot |E|^2)$ begrenzt werden kann.

Das Argument geht wie folgt. Für jeden Knoten $v \in V$ sei $l(v)$ sein minimaler Abstand von der Quelle s. Der dem Graphen G zugeordnete *Schichtengraph* $L(G)$ ist derjenige Teilgraph von G, der nur solche Kanten $(u, v) \in E$ enthält, für die $l(v) = l(u) + 1$ gilt. Es ist klar, dass $L(G)$ für jeden Knoten $v \in V$ alle Pfade von s nach v minimaler Länge enthält.

Betrachten wir nun einen Erweiterungspfad p minimaler Länge in G_f; das heißt, p wurde gemäß der Edmonds-Karp Strategie gefunden. Dann liegen die Kanten von p in $L(G_f)$. Nun wird der Fluss entlang des Pfades p um $c_f(p)$ erhöht. Das nächste Restnetzwerk G' nach dieser Flusserhöhung unterscheidet sich von G_f dadurch, dass mindestens eine Kante (u, v) auf p nun verschwunden ist (diejenige mit $c_f(u, v) = c_f(p)$). Ferner können einige weitere Kanten hinzugekommen sein; und zwar sind dies Kanten (v, u), wobei (u, v) auf p liegt. Diese neuen Kanten sind also „rückwärtsgerichtet“, denn $l(u) = l(v) -$

1.

Betrachten wir nun den nächsten Erweiterungspfad q in G'. Falls dieser auch in $L(G_f)$ vorkommt, so muss dieser dieselbe Länge wie p haben. Falls dieser dagegen nicht in $L(G_f)$ vorkommt, dann dieser Pfad nicht minimale Länge haben. Daher muss in diesem Fall die Länge von q mindestens um 1 größer sein als die Länge von p.

Diese Überlegung zeigt uns, dass die Pfadlängen der Erweiterungspfade in jedem Schleifendurchlauf entweder gleichbleiben, oder echt zunehmen. Die Pfadlängen können aber höchstens für $2|E| = \mathcal{O}(|E|)$ viele Schritte gleichbleiben, denn G_f (und damit auch $L(G_f)$) hat zu Beginn höchstens $2|E|$ viele Kanten, und nach jedem Auffinden eines Erweiterungspfades wird mindestens eine Kante entfernt. Dass G_f höchstens $2|E|$ viele Kanten hat, ergibt sich daraus, dass jede Kante (u, v) in G Anlass zu höchstens zwei Kanten in G_f gibt (nämlich (u, v) und (v, u)).

Als mögliche Pfadlängen kommen nur die Zahlen $1, 2, \ldots, |V| - 1$ in Frage. Zusammengefasst heißt dies, dass nicht mehr als

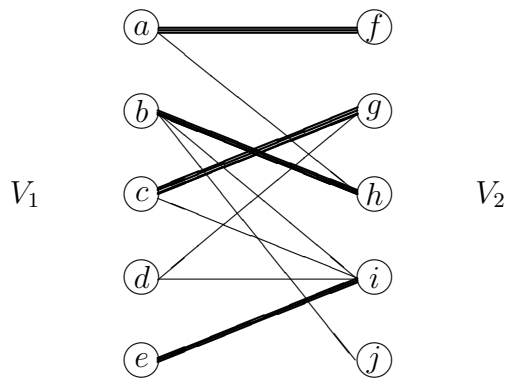
$$\begin{aligned}&\text{(Anzahl möglicher Pfadlängen)}\\ &\cdot \text{(Anzahl Erweiterungspfade mit derselben Länge)}\\ &= \mathcal{O}(|V| \cdot |E|)\end{aligned}$$

viele Schleifendurchläufe möglich sind.

6.6 Maximales Matching

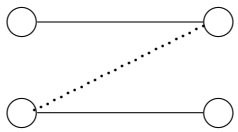
Gegeben sei ein bipartiter Graph. Es geht um die Aufgabe, ein möglichst großes *Matching* zu bestimmen. Hierbei ist ein Matching M eine Teilmenge der Kantenmenge, $M \subseteq E$, so dass keine zwei Kanten in M einen gemeinsamen Knoten besitzen.

Beispiel:
Die fett gezeichneten Kanten charakterisieren ein Matching der Größe 4 in dem folgenden bipartiten Graphen $G = (V_1 + V_2, E)$. Dieses Matching ist maximal, denn es gibt in diesem Fall kein Matching der Größe 5. Warum das bei diesem Beispiel so ist, beantwortet der sog. *Heiratssatz*, der etwas weiter unten bewiesen wird.



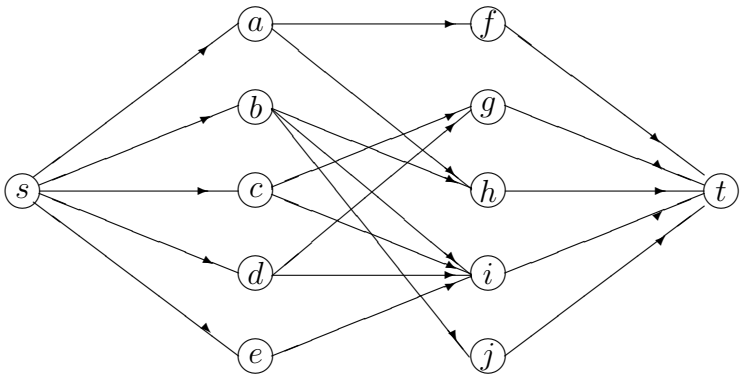
Dieses Matching-Problem ist nicht trivial; zum Beispiel funktioniert der Greedy-Ansatz nicht: wenn wir einfach nur Kanten auswählen, solange sie die Matching-Bedingung erfüllen, bis keine weitere Kante mehr hinzugenommen werden kann, so entsteht am Ende nicht unbedingt ein maximales Matching. Formaler: die Menge aller möglichen Matchings eines gegebenen bipartiten Graphen ist im Allgemeinen kein Matroid, und daher funktioniert aufgrund der in Kapitel 5 entwickelten Theorie der Greedy-Ansatz nicht.

Beispiel:
Gegeben sei der folgende Graph mit drei Kanten.



Sei A die einelementige Kantenmenge, die nur aus der gestrichelten Kante besteht; sei B die Kantenmenge, die aus den beiden durchgezogenen Kanten besteht. Für diese Wahl von A und B ist das Austauschaxiom verletzt, denn die partielle Lösung A kann nicht durch Hinzunahme einer Kante aus B zu einem größeren Matching erweitert werden. Mit anderen Worten, sollte der Greedy-Algorithmus mal mit der gestrichelten Kante begonnen haben, so ist er in eine „Sackgasse“ geraten, und kann die optimale Lösung, die aus den beiden durchgezogenen Kanten besteht, nicht mehr erreichen, da er einmal getroffene Entscheidungen nicht mehr rückgängig machen kann.

Wir können dieses Matching-Problem aber mit Hilfe des Ford-Fulkerson-Algorithmus lösen, indem wir links eine Quelle s und rechts eine Senke t hinzufügen, die Quelle mit allen Knoten in V_1 und t mit allen Knoten in V_2 verbinden und den Graph (von links nach rechts) gerichtet machen. Wir führen dies bei dem obigen Beispielgraphen durch.



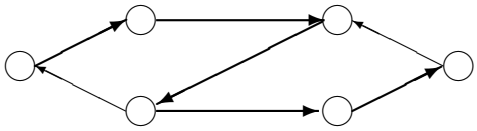
Allen Kanten ordnen wir die Kapazität 1 zu. Wir wenden nun den Algorithmus von Ford-Fulkerson auf dieses Netzwerk an. Das Ergebnis ist ein maximaler Fluss f. Da die c-Werte alle gleich 1 sind, können die f-Werte an den Kanten nur 0 oder 1 sein. Das gesuchte Matching wird gebildet durch diejenigen Kanten zwischen V_1 und V_2 mit dem f-Wert gleich 1.

Diese Kantenmenge bildet tatsächlich ein Matching, denn in jeden Knoten links fließt maximal ein Fluss von 1 hinein, deshalb kann auch nur maximal eine hinausgehende Kante den f-Wert 1 erhalten. Analog ist es auf der rechten Seite: von jedem rechten Knoten fließt ein Fluss von höchstens 1 in Richtung t hinaus, also kann es auch nur höchstens eine hereinkommende Kante mit $f = 1$ geben.

Dieses Matching ist tatsächlich maximal, da f maximal ist. Wenn es ein Matching mit mehr Kanten gäbe, dann hätte der diesem Matching zugeordnete Fluss einen höheren Betrag, was ein Widerspruch ist, denn Ford-Fulkerson bestimmt nachgewiesenermaßen immer einen *maximalen* Fluss.

Wir analysieren nochmals die Komplexität von Ford-Fulkerson in der hier betrachteten speziellen Anwendung. Da hier $|f^*| = \mathcal{O}(|V|)$ gilt, hat das Verfahren die Komplexität $\mathcal{O}(|V| \cdot |E|)$.

Betrachten wir nochmals das obige Beispiel zum Scheitern des Greedy-Ansatzes. Beim Ford-Fulkerson-Algorithmus (selbst mit der Edmonds-Karp-Strategie) kann es ebenso passieren, dass im ersten Schritt ein Erweiterungsfad gewählt wurde, der über die gestrichelte Kante läuft. Wie kommt Ford-Fulkerson aus dieser „Sackgasse“ wieder heraus? Das zugehörige Restnetzwerk nach Auswahl der gestrichelten Kante sieht wie folgt aus:



Die fetten Linien deuten einen Erweiterungspfad an, welcher sozusagen die im ersten Schritt gemachte Entscheidung wieder rückgängig macht und stattdessen die beiden anderen Kanten auswählt.

Der folgende Satz gibt ein Kriterium an, wann ein bipartiter Graph ein maximales Matching besitzt.

SATZ („Heiratssatz“ von Hall):
Sei $G = (V, E)$ mit $V = V_1 + V_2$ ein bipartiter Graph. Es gibt ein Matching $M \subseteq E$ mit $|M| = |V_1|$ genau dann, wenn für alle Teilmengen $A \subseteq V_1$ gilt, dass die Anzahl der Nachbarn der Knoten in A mindestens genauso groß ist wie $|A|$ (formal: $|N(A)| \geq |A|$).

Beispiel:
Der bipartite Graph auf Seite 97 besitzt kein Matching mit 5 Kanten, da die drei Knoten c, d, e nur die zwei Nachbarn g, i besitzen, also $|N(\{c, d, e\})| = |\{g, i\}| = 2$.

Beweis:
Die eine Richtung des Satzes ist klar: Wenn ein Matching M mit $|M| = |V_1|$ existiert, so hat jeder Knoten links einen „Partner“ auf der rechten Seite. Also hat jede Teilmenge A links mindestens $|A|$ Nachbarn auf der rechten Seite.

Für die Umkehrung verwenden wir den Ford-Fulkerson-Algorithmus bzw. dessen Korrektheit, welche durch das Min-Cut-Max-Flow-Theorem zu Stande kommt. Es gelte $|A| \leq |N(A)|$ für alle Teilmengen $A \subseteq V_1$. Sei f ein mittels Ford-Fulkerson (wie oben

beschrieben) ermittelter maximaler Fluss. Dieser Fluss wird in dem G zugeordneten gerichteten Graphen G' bestimmt. Sei X die Menge aller Knoten, die sich im zuletzt erhaltenen Restnetzwerk zu G' von der Quelle s aus erreichen lassen. Nach dem Min-Cut-Max-Flow-Theorem gilt für den betreffenden Schnitt $(X, \overline{X})$, dass $c(X, \overline{X}) = |f|$. Sofern X einen Knoten aus V_1 enthält, so wurde dieser nicht beim maximalen Matching (das zu dem maximalen Fluss korrespondiert) verwendet. Daher sind die V_2-Nachbarn eines solchen Knotens ebenfalls erreichbar, also in X enthalten, formal: $N(X \cap V_1) \subseteq X$. Unter Verwenden der Voraussetzung auf $X \cap V_1$ erhalten wir daher:

$$|f| = c(X, \overline{X}) = |V_1 - X| + |N(X \cap V_1)| \geq |V_1 - X| + |X \cap V_1| = |V_1|$$

Daher hat der maximale Fluss f (der einem maximalen Matching entspricht) den Betrag $|f| = |V_1|$. □

Kapitel 7

Algebraische und zahlentheoretische Algorithmen

Wir stellen einige bekannte Algorithmen aus dem Bereich der Algebra und Zahlentheorie vor. Die zahlentheoretischen Algorithmen haben ihren hauptsächlichen Anwendungsbereich in der Kryptographie, zum Beispiel beim sog. RSA-Verfahren.

7.1 Multiplikation großer Zahlen

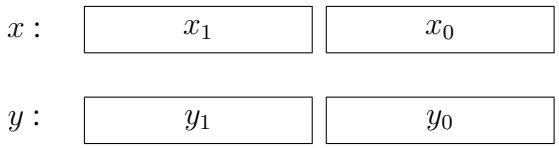
Große Zahlen können wir leicht nach der *Schulmethode* miteinander multiplizieren.

Beispiel:

1	2	3	4	5	$*$	6	7	8	9	0
		7	4	0	7	0				
			8	6	4	1	5			
				9	8	7	6	0		
				1	1	1	1	0	5	
+						0	0	0	0	0
		8	3	8	1	0	2	0	5	0

Es ist klar, dass diese Methode die Bit-Komplexität $\Theta(n^2)$ hat, denn die erzeugte Tabelle hat diese Größenordnung, wobei wir hier von der Bit-Komplexität ausgehen. Jede Operation gemäß des „kleinen 1×1“ hat konstante Komplexität.

Karatsuba und Ofman haben eine schnellere Methode angegeben, eine klassische *divide-and-conquer*-Methode. Und zwar zerlegen wir die Ziffernfolgen der zu multiplizierenden Zahlen x und y in zwei gleich lange Teile wie folgt:



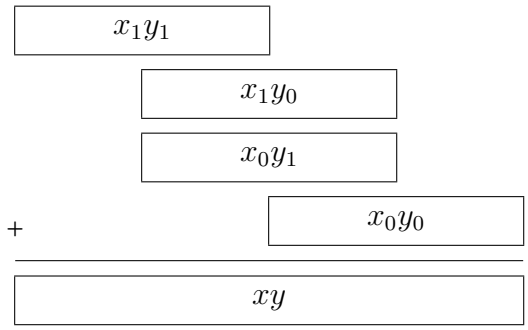
Das heißt, x und y haben die Darstellung:

$$x = x_1 b^{n/2} + x_0, \quad y = y_1 b^{n/2} + y_0$$

wobei b die verwendete Basis ist (z.B. $b = 10$ oder $b = 2$).
Wenn wir x und y multiplizieren, erhalten wir:

$$\begin{aligned} x \cdot y &= (x_1 b^{n/2} + x_0) \cdot (y_1 b^{n/2} + y_0) \\ &= x_1 y_1 \cdot b^n + (x_1 y_0 + x_0 y_1) \cdot b^{n/2} + x_0 y_0 \end{aligned}$$

Das bedeutet, dass wir die Aufgabe, zwei n-stellige Zahlen miteinander zu multiplizieren, darauf reduziert haben, 4 Multiplikationen von $(n/2)$-stelligen Zahlen (rekursiv) durchzuführen, und die Ergebnisse dann gemäß obiger Formel (nach vorherigen „Shifts“ um $n/2$ bzw. um n Stellen) zu addieren.
Skizze:

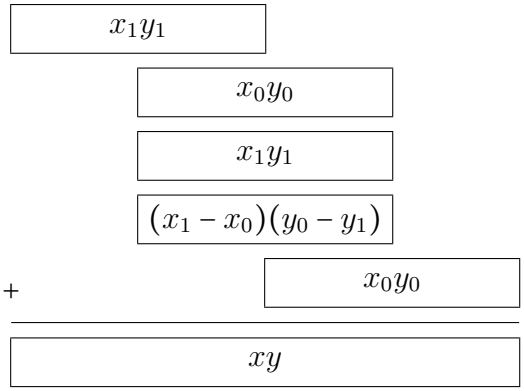


Die Komplexität $T(n)$ dieses Vorgehens können wir abschätzen durch $T(n) = 4T(n/2) + \Theta(n)$, was nach dem „Master-Theorem“ (Seite 17) die Komplexität $\Theta(n^{\log_2 4}) = \Theta(n^2)$ hat. Hoppla – wir haben gegenüber der Schulmethode also gar nichts gewonnen?! Tatsächlich haben wir uns eher einen noch größeren Verwaltungsaufwand durch die rekursiven Aufrufe eingehandelt.

An dem Master-Theorem können wir aber auch sehen, dass wir Komplexität einsparen könnten, wenn es gelänge, die Berechnung statt mit 4 nur mit 3 rekursiven Aufrufen durchzuführen. Genau das ist möglich, wenn wir den gemischten Term $x_1y_0 + x_0y_1$ umschreiben in $x_1y_1 + x_0y_0 - (x_0 - x_1)(y_0 - y_1)$. Dies ergibt die Darstellung

$$x \cdot y = x_1 y_1 \cdot b^n + (x_1 y_1 + x_0 y_0 + (x_1 - x_0)(y_0 - y_1)) \cdot b^{n/2} + x_0 y_0$$

Bei dieser Darstellung müssen tatsächlich nur 3 Multiplikationen von $(n/2)$-stelligen Zahlen durchgeführt werden, nämlich x_1y_1, x_0y_0 und $(x_1 - x_0)(y_0 - y_1)$. Die durch die 3 rekursiven Aufrufe erhaltenen Ergebnisse müssen dann mittels geeigneter „Shifts“ zusammenaddiert werden:



Damit ergibt sich (mit Hilfe des Master-Theorems) eine Komplexität von

$$T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{\log_2 3}), \text{ wobei } \log_2 3 \approx 1.585.$$

7.2 Schnelle Matrizenmultiplikation nach Strassen

Wie schnell kann man zwei $n \times n$ Matrizen miteinander multiplizieren? Wie viele Elementaroperationen (Multiplikationen und Additionen von einzelnen Zahlen) benötigt man dazu?
Sei

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nn} \end{pmatrix}$$

Dann gilt:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Wenn wir die Berechnung der Produktmatrix direkt anhand der Definition durchführen, so haben wir es mit n^3 Einzel-Multiplikationen und $n^2(n-1)$ Einzel-Additionen zu tun.
Betrachten wir den Fall von 2×2 Matrizen:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Laut angegebener Formeln brauchen wir hier 8 Multiplikationen und 4 Additionen. Man kann dasselbe Ergebnis aber auch durch 7 Multiplikationen und 18 Additionen (bzw.

Subtraktionen) erhalten, indem man folgende Berechnungen durchführt:

$$\begin{aligned} I &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}) \\ II &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \\ III &= (a_{11} - a_{21}) \cdot (b_{11} + b_{12}) \\ IV &= (a_{11} + a_{12}) \cdot b_{22} \\ V &= a_{11} \cdot (b_{12} - b_{22}) \\ VI &= a_{22} \cdot (b_{21} - b_{11}) \\ VII &= (a_{21} + a_{22}) \cdot b_{11} \\ \\ c_{11} &= I + II - IV + VI \\ c_{12} &= IV + V \\ c_{21} &= VI + VII \\ c_{22} &= II - III + V - VII \end{aligned}$$

Wieso sollten 7 Multiplikationen und 18 Additionen besser sein als 8 Multiplikationen und 4 Additionen? Entscheidend sind die Multiplikationen, denn über die Multiplikationen können wir ein rekursives divide-and-conquer Verfahren laufen lassen. Hierzu verlassen wir jetzt die 2×2 Matrizen und betrachten wieder allgmeine $n \times n$ Matrizen (hierbei sei n eine Zweierpotenz). Wir zerlegen alle beteiligten Matrizen in 4 $(n/2) \times (n/2)$ Matrizen und erhalten folgende Darstellung:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

Wir können genau dieselben Formeln wie oben verwenden, nur dass wir diese nun als Multiplikationen bzw. Additionen von $(n/2) \times (n/2)$ Matrizen verstehen. Die Additionen/Subtraktionen sind hierbei komponentenweise zu verstehen, und die 7 Multiplikationen führen wir durch rekursive Aufrufe des Matrizen-Multiplikationsalgorithmus durch. Damit erhalten wir die folgende rekursive Prozedur.

Algorithmus 7.1 `matProd`(n, A, B)

1: **IF** $n = 1$ **THEN**
2: `return` (AB)
3: **ELSE**
4: Berechne $I, \ldots, VII, C_{11}, C_{12}, C_{21}, C_{22}$ gemäß obiger Formeln
5: Die 7 vorkommenden Multiplikationen werden durch 7 rekursive Aufrufe der Form `matProd`$(\frac{n}{2}, \ldots)$ erledigt
6: `return` $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Sei $T_M(n)$ bzw. $T_A(n)$ die Anzahl der elementaren Multiplikationen bzw. Additionen, die beim Aufruf von *MatProd*$(n, \ldots)$ auftreten. Es gilt:

$$T_M(n) = \begin{cases} 1, & n = 1 \\ 7 \cdot T_M(n/2), & n > 1 \end{cases}$$

Es ergibt sich

$$T_M(n) = \underbrace{7 \cdot 7 \cdots 7}_{(\log n)\text{-mal}} = 7^{\log n} = n^{\log 7} \leq n^{2.8074}$$

Ferner haben wir:

$$T_A(n) = \begin{cases} 0, & n = 1 \\ 18(n/2)(n/2) + 7T_A(n/2), & n > 1 \end{cases}$$

Aus dem Master-Theorem (Seite 17) ergibt sich sofort $T_A(n) = \mathcal{O}(n^{2.8074})$. Somit haben wir auch insgesamt die Komplexität $\mathcal{O}(n^{2.8074})$ – im Vergleich zur Komplexität $\Theta(n^3)$, wenn wir nach der Schulmethode multiplizieren.

Es ist anzunehmen, dass der versteckte konstante Faktor bei $\mathcal{O}(n^{2.8074})$ in diesem Fall größer ist als derjenige bei $\Theta(n^3)$. Das heißt, das Verfahren wird sich erst bei Matrizen ab einer gewissen Mindestgröße n_0 (zum Beispiel $n_0 = 8$) lohnen. Um dieser Tatsache Rechnung zu tragen, können wir die obige Prozedur *MatProd* noch modifizieren. Statt

IF $n = 1$ **THEN**
  `return` (AB)

schreiben wir

IF $n < n_0$ **THEN**
  Berechne $C = AB$ nach der Schulmethode
  `return` C

Wir merken noch an, dass in der Zwischenzeit immer wieder neue Algorithmen für die Matrizenmultiplikation vorgeschlagen wurden; der „Beste“ davon (von Coppersmith und Winograd, 1986) erreicht die Komplexität $\mathcal{O}(n^{2.376})$. Allerdings ist die in der O-Notation versteckte Konstante so groß, dass dieser Algorithmus nicht praktikabel ist.

7.3 Polynommultiplikation und die FFT

Wir betrachten die Aufgabe, zwei Polynome zu multiplizieren. Die Polynome seien gegeben in ihrer Koeffizientendarstellung. Sei das Polynom

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

und das Polynom

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

gegeben durch die Koeffizienten-Tupel $a = (a_0, \ldots, a_{n-1})$ und $b = (b_0, \ldots, b_{n-1})$. (Sollte eines der Polynome weniger als n Koeffizienten haben, so muss entsprechend mit Nullen aufgefüllt werden. Später wird sich herausstellen, dass es günstig ist, wenn n eine Zweierpotenz ist. In diesem Fall gilt dieselbe Bemerkung: dann muss bis zur nächsten Zweierpotenz mit Nullen aufgefüllt werden).

Das Produktpolynom $C(x) = A(x) \cdot B(x)$ ergibt sich zu

$$C(x) = \left(\sum_{j=0}^{n-1} a_j x^j\right) \cdot \left(\sum_{k=0}^{n-1} b_k x^k\right) = \sum_{i=0}^{2n-2} c_i x^i$$

wobei

$$c_i = \sum_{k=0}^{i} a_k b_{i-k}$$

Diese letzte Operation, mittels der die Koeffizienten c_i zu bestimmen sind, nennt man auch oft *Faltung* (engl.: convolution).

Wenn man die Polynommultiplikation entsprechend dieser Formeln durchführt, so sind also $2n-1$ Koeffizienten zu bestimmen, wobei jeder von diesen eine Faltung erfordert. Das macht zusammen die Komplexität $\mathcal{O}(n^2)$. (Die Komplexitätsangaben sind im uniformen Kostenmaß zu verstehen, d.h. jede elementare Multiplikation bzw. Addition hat die Komplexität $\mathcal{O}(1)$).

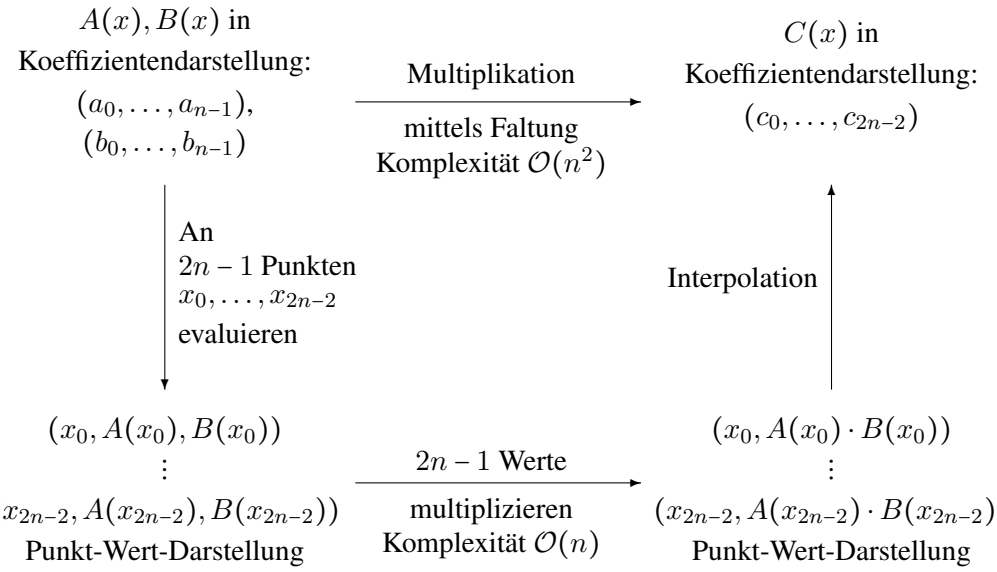
Eine mögliche effizientere Implementierung ergibt sich, wenn man die Idee der schnellen Multiplikation ganzer Zahlen aus Abschnitt 7.1 wieder aufgreift: Wir zerlegen das Polynom A, gegeben durch die Folge seiner n Koeffizienten $(a_0, \ldots, a_{n-1})$ in zwei Polynome mit jeweils $n/2$ vielen Koeffizienten. Das Polynom A_0 ist hierbei gegeben durch $(a_0, \ldots, a_{n/2-1})$ und das Polynom A_1 ist gegeben durch $(a_{n/2}, \ldots, a_{n-1})$. Es gilt $A(x) = A_0(x) + x^{n/2} A_1(x)$. Analog kann man das Polynom B in B_0 und B_1 zerlegen. Für das Produktpolynom C ergibt sich:

$$\begin{aligned}
C(x) &= A(x)B(x) \\
&= (A_0(x) + x^{n/2}A_1(x))(B_0(x) + x^{n/2}B_1(x)) \\
&= A_0(x)B_0(x) + x^{n/2}(A_0(x)B_1(x) + A_1(x)B_0(x)) + x^n A_1(x)B_1(x) \\
&= A_0(x)B_0(x) + x^n A_1(x)B_1(x) \\
&\quad + x^{n/2}(A_0(x)B_0(x) + A_1(x)B_1(x) + (A_1(x) - A_0(x)(B_0(x) - B_1(x)))
\end{aligned}$$

Man erkennt, dass man die Multiplikation in divide-and-conquer Art programmieren kann (vgl. Abschnitt 7.1), wobei drei rekursive Aufrufe notwendig sind. Diese rekursiven Aufrufe beziehen sich auf die folgenden Produkte von Polynomen mit je $(n/2)$-vielen Koeffizienten: $A_0(x)B_0(x)$, $A_1(x)B_1(x)$ und $(A_0(x) - A_1(x)(B_0(x) - B_1(x))$. Unter der Annahme, dass alle Elementar-Additionen und Multiplikationen in der Zeit $\mathcal{O}(1)$ ausführbar sind, erhalten wir die Rekursionsgleichung $T(n) = 3T(n/2) + \Theta(n)$, welche mit Hilfe des Master-Theorems (Seite 17) die Lösung $\mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.585})$ hat.

Das folgende Schema skizziert einen weiteren möglichen Weg, von den beiden Polynomen $A(x)$ und $B(x)$ (in Koeffizientendarstellung) zum Produktpolynom $C(x) = A(x) \cdot B(x)$ (in Koeffizientendarstellung) zu gelangen. Hierbei ist ein „Umweg“ skizziert, der auch zum Produktpolynom $C(x)$ führt, allerdings zunächst die Polynome $A(x)$ und $B(x)$ an beliebigen $2n-1$ Punkten (Stützstellen) $x_0, \ldots, x_{2n-2}$ evaluiert zur *Punkt-Wert-Darstellung*. Danach werden die Werte Punkt für Punkt miteinander multipliziert

und schließlich wird aus den $2n-1$ vielen Werten die Koeffizienten des Produktpolynoms rekonstruiert. (Diesen Vorgang nennt man *Interpolation*).



Wenn man das Evaluieren an beliebigen $2n-1$ Stützstellen mittels *Hornerschema* implementiert – also gemäß der Formel

$$A(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \cdots + a_1) \cdot x + a_0$$

und die Interpolation nach Newton, so ergibt jede dieser Operationen eine Komplexität von $\mathcal{O}(n^2)$. Man kann solcherart also nichts an Effizienz gegenüber der direkten Methode gewinnen.

Wir können aber ausnützen, dass wir völlige Wahlfreiheit bei den Stützstellen haben. Wenn wir diese geschickt wählen, so kann das Evaluieren (ebenso wie das Interpolieren) an den Stellen $x_0, \ldots, x_{2n-2}$ effizienter als mit Komplexität $\mathcal{O}(n^2)$ (nämlich mit Komplexität $\mathcal{O}(n \log n)$) implementiert werden.

Die *diskrete Fourier-Transformation* (kurz: DFT) besteht darin, ein Polynom, das durch n Koeffizienten $(a_0, \ldots, a_{n-1})$ gegeben ist, an genau n Stützstellen, und zwar den sog. *n-ten Einheitswurzeln* auszuwerten. Hier müssen also die Anzahl der Koeffizienten des Polynoms und die Anzahl der Stützstellen übereinstimmen. In der obigen Situation hat das Polynom n Koeffizienten und soll an $2n-1$ Stellen evaluiert werden. Indem wir die Koeffizientendarstellung des Polynoms mit Nullen auffüllen, können wir die Anzahl der Koeffizienten gleich der Anzahl der Stützstellen machen.

In der folgenden Diskussion sei der Einfachheit halber n=Anzahl Koeffizienten=Anzahl der Stützstellen. Außerdem sei n eine Zweierpotenz.

Der Trick besteht, wie gesagt, darin, als Evaluationspunkte die (komplexen) *n-ten Einheitswurzeln* zu wählen. Ein komplexe Zahl x ist n-te Einheitswurzel, falls $x^n = 1$. Es gibt im Komplexen genau n Lösungen der Gleichung $x^n = 1$. Dieses sind die Zahlen $x_{0,n}, \ldots, x_{n-1,n}$, wobei

$$x_{k,n} = e^{ik2\pi/n}$$

Wir erinnern in diesem Zusammenhang an die *Eulersche Formel*:

$$e^{i\phi} = \cos(\phi) + i\sin(\phi)$$

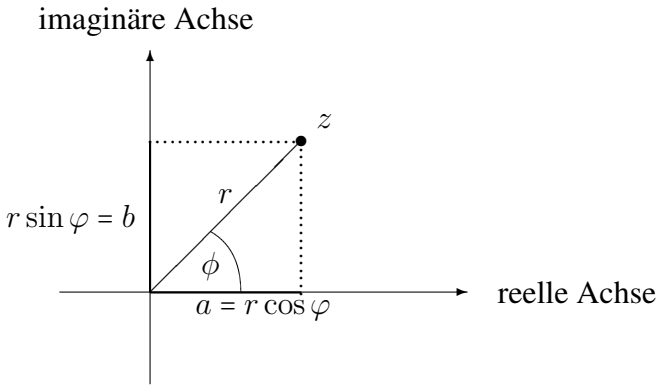
bzw.

$$sin(\phi) = \frac{e^{i\phi} - e^{-i\phi}}{2i} \quad \text{und} \quad cos(\phi) = \frac{e^{i\phi} + e^{-i\phi}}{2}$$

Hierbei ist i die imaginäre Einheit mit $i^2 = -1$.
Eine komplexe Zahl z lässt sich darstellen:

- *algebraisch* durch eine Zerlegung in Real- und Imaginärteil: $z = a + ib$,
- *polar* durch Angabe eines Radius r und eines Winkels ϕ als $z = r\cos(\phi) + ir\sin(\phi)$,
- oder *exponentiell* durch $z = re^{i\phi}$.

Veranschaulichung in der *Gaußschen Zahlenebene*:

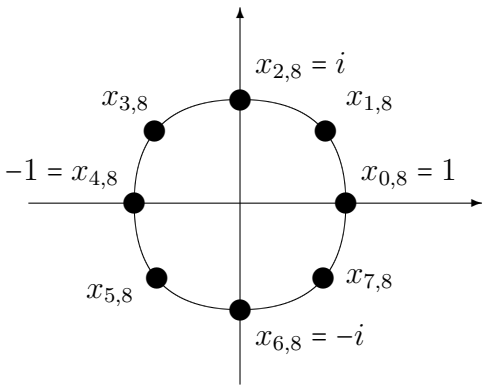


Im Folgenden arbeiten wir mit der exponentiellen Darstellung, da hier die Multiplikation besonders einfach geht:

$$re^{i\phi} \cdot r'e^{i\phi'} = (rr')e^{i(\phi+\phi')}$$

Das heißt, die Radien müssen miteinander multipliziert werden und die Winkel müssen addiert werden.

Die n-ten Einheitswuzeln liegen in der Gaußschen Zahlenebene auf dem Einheitskreis gleichmäßig verteilt. Das folgende Bild zeigt den Fall $n = 8$:



Überprüfen wir, dass $x_{k,n}$ tatsächlich eine n-te Einheitswurzel ist:

$$(x_{k,n})^n = (e^{ik2\pi/n})^n = e^{ik2\pi} = \cos(k2\pi) + i\sin(k2\pi) = 1 + i \cdot 0 = 1$$

Außerdem ergibt sich, dass man alle n-ten Einheitswurzeln als Potenzen von $x_{1,n}$ erhalten kann. Sei im Folgenden $z := x_{1,n} = e^{i2\pi/n}$. Dann gilt:

$$(x_{0,n}, x_{1,n}, \ldots, x_{n-1,n}) = (z^0, z^1, \ldots, z^{n-1})$$

Eine weitere wichtige Beobachtung ist, dass man durch Quadrieren der n-ten Einheitswurzeln gerade die $(n/2)$-ten Einheitswurzeln (und zwar zweimal) erhält. Sei im Folgenden $k = 0, \ldots, (n/2) - 1$:

$$(x_{k,n})^2 = e^{ik2\pi/(n/2)} = x_{k,n/2}$$

$$(x_{(n/2)+k,n})^2 = e^{i((n/2)+k)2\pi/(n/2)} = e^{i2\pi} \cdot e^{k2\pi/(n/2)} = e^{k2\pi/(n/2)} = x_{k,n/2}$$

DEFINITION (DFT):
Zu gegebenem Koeffizientenvektor $a = (a_0, a_1, \ldots, a_{n-1})$ (welcher das Polynom A repäsentiert) bezeichnet der Vektor $y = (y_0, y_1, \ldots, y_{n-1})$ mit

$$\begin{aligned} y_k &= A(x_{k,n}) \\ &= \sum_{j=0}^{n-1} a_j e^{ijk2\pi/n} \end{aligned}$$

die diskrete Fourier-Transformierte *(DFT) von a.*

Das heißt, y zu berechnen bedeutet das Evaluieren von A an den n-ten Einheitswurzeln. Es wird sich als nützlich erweisen, das Polynom A in zwei kleinere Polynome mit jeweils $n/2$ vielen Koeffizienten aufzuspalten. Sei

$$\begin{aligned} A^{\text{even}}(x) &:= \sum_{k=0}^{(n/2)-1} a_{2k} x^k \\ A^{\text{odd}}(x) &:= \sum_{k=0}^{(n/2)-1} a_{2k+1} x^k \end{aligned}$$

Das heißt, die Polynome A^{even} und A^{odd} besitzen jeweils $n/2$ Koeffizienten und A^{even} wird repräsentiert durch den Koeffizientenvektor

$$(a_0, a_2, a_4, \ldots, a_{n-2})$$

und A^{odd} wird repräsentiert durch

$$(a_1, a_3, a_5, \ldots, a_{n-1}).$$

Es gilt die Beziehung:

$$A(x) = A^{\text{even}}(x^2) + x \cdot A^{\text{odd}}(x^2)$$

In dieser Formel und in der obigen Beobachtung, dass beim Quadrieren der n-ten Einheitswurzeln gerade die $(n/2)$-ten Einheitswurzeln entstehen, liegt der Schlüssel für den *FFT-Algorithmus* (FFT=fast Fourier transform). (Das heißt, der FFT ist ein schneller Algorithmus zur Berechnung der DFT).

Wir beschreiben den Algorithmus zunächst verbal, und dann formal als Programm. Gegeben ist also der Koeffizientenvektor $a = (a_0, \ldots, a_{n-1})$, der das Polynom A repräsentiert. Wir zerlegen diesen zunächst in die beiden Vektoren $a^{\text{even}} = (a_0, a_2, a_4, \ldots, a_{n-2})$ und $a^{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, jeweils der Länge $n/2$. Dann wenden wir den FFT *rekursiv* auf a^{even} und auf a^{odd} an und erhalten die Fourier-Transformierten

$$y^{\text{even}} = (y_0^{\text{even}}, y_1^{\text{even}}, \ldots, y_{(n/2)-1}^{\text{even}}) = (A^{\text{even}}(x_{0,n/2}), \ldots, A^{\text{even}}(x_{(n/2)-1,n/2}))$$

und

$$y^{\text{odd}} = (y_0^{\text{odd}}, y_1^{\text{odd}}, \ldots, y_{(n/2)-1}^{\text{odd}}) = (A^{\text{odd}}(x_{0,n/2}), \ldots, A^{\text{odd}}(x_{(n/2)-1,n/2}))$$

hiervon. Aus der obigen Formel lassen sich nun die gesuchten Werte

$$(y_0, \ldots, y_{n-1}) = (A(x_{0,n}), \ldots, A(x_{n-1,n}))$$

rekonstruieren: Für $k = 0, 1, \ldots, (n/2) - 1$ gilt:

$$\begin{aligned} y_k &= A(x_{k,n}) \\ &= A^{\text{even}}((x_{k,n})^2) + x_{k,n} \cdot A^{\text{odd}}((x_{k,n})^2) \\ &= A^{\text{even}}(x_{k,n/2}) + x_{k,n} \cdot A^{\text{odd}}(x_{k,n/2}) \\ &= y_k^{\text{even}} + x_{k,n} \cdot y_k^{\text{odd}} \end{aligned}$$

und entsprechend

$$\begin{aligned} y_{(n/2)+k} &= A(x_{(n/2)+k,n}) \\ &= A^{\text{even}}((x_{(n/2)+k,n})^2) + x_{(n/2)+k,n} \cdot A^{\text{odd}}((x_{(n/2)+k,n})^2) \\ &= A^{\text{even}}(x_{k,n/2}) + x_{(n/2)+k,n} \cdot A^{\text{odd}}(x_{k,n/2}) \\ &= y_k^{\text{even}} - x_{k,n} \cdot y_k^{\text{odd}} \end{aligned}$$

Die Umformung $x_{(n/2)+k,n} = -x_{k,n}$ ist begründet durch die Beobachtung $x_{n/2,n} = -1$.
Nun lässt sich der FFT-Algorithmus wie folgt formulieren:

Algorithmus 7.2 `FFT`$(n, (a_0, \ldots, a_{n-1})$

```
1:  IF n = 1 THEN
2:     return (a_0)
3:  ELSE
4:     a^even = (a_0, a_2, ..., a_{n-2})
5:     a^odd = (a_1, a_3, ..., a_{n-1})
6:     y^even = FFT(n/2, a^even)
7:     y^odd = FFT(n/2, a^odd)
8:     x = 1
9:     z = e^{i2π/n}
10:    FOR k = 0 TO (n/2) − 1 DO
11:       y_k = y_k^even + x · y_k^odd
12:       y_{(n/2)+k} = y_k^even − x · y_k^odd
13:       x = x · z
14:    return (y_0, y_1, ..., y_{n-1})
```

Die Komplexität dieses Algorithmus ergibt sich aus der Rekursionsgleichung $T(n) = 2T(n/2) + \Theta(n)$, deren Lösung nach Master-Theorem (Seite 17) bekanntermaßen $T(n) = \Theta(n \log n)$ ist.

Die *inverse DFT* unterscheidet sich von der DFT nur geringfügig: Um aus einem gegebenen Wertevektor $(y_0, \ldots, y_{n-1})$ die Koeffizienten $(a_0, \ldots, a_{n-1})$ zurückzuerhalten, wende man die folgende Formel an:

$$a_k = \frac{1}{n} \cdot \sum_{j=0}^{n-1} y_j e^{-ijk2\pi/n}$$

Das heißt, dass die inverse DFT auf die DFT zurückgeführt werden kann, und damit unter Zuhilfenahme des FFT-Algorithmus – ebenfalls mit der Komplexität $\Theta(n \log n)$ – berechnet werden kann. Symbolisch können wir dies wie folgt ausdrücken. Sei *DFT* ein Operator, der den Koeffizientenvektor a in den Wertevektor y überführt; und $\textit{DFT}^{-1}$ überführt umgekehrt y nach a. Dann gilt $\textit{DFT}^{-1}(y) = \textit{konj}(\textit{DFT}(y))/n$, wobei *konj* den Übergang zu den konjugiert komplexen Zahlen und „$/n$“ die komponentenweise Division durch n bedeutet.

Seien nun zwei Polynome mit Koeffizientenvektoren a und b gegeben. Diese Vektoren seien mit genügend Nullen aufgefüllt. Dann gilt für den Koeffizientenvektor c des Produktpolynoms

$$c = \textit{DFT}^{-1}(\textit{DFT}(a) * \textit{DFT}(b))$$

Hierbei bedeutet $*$ die komponentenweise Multiplikation der beiden Vektoren, also

$$(y_0, \ldots, y_{n-1}) * (y'_0, \ldots, y'_{n-1}) = (y_0 y'_0, \ldots, y_{n-1} y'_{n-1})$$

Zur Begründung für die obige Formel zur inversen DFT beobachten wir, dass sich die DFT äquivalent durch eine Vektor-Matrix-Multiplikation beschreiben lässt. Hierzu sei

wieder $z = e^{i2\pi/n}$:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & z & z^2 & \cdots & z^{n-1} \\ 1 & z^2 & z^4 & \cdots & z^{2(n-1)} \\ 1 & z^3 & z^6 & \cdots & z^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z^{n-1} & z^{2(n-1)} & \cdots & z^{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Diese Matrix wird *Vandermondesche Matrix* genannt. Obige Formel für die inverse DFT behauptet, dass die zur Vandermondeschen Matrix inverse Matrix gerade so aussieht, dass im (j,k)-ten Eintrag z^{-jk}/n steht. Wir überprüfen dies indem wir die Vandermondesche Matrix mit der (behaupteten) Inversen multiplizieren. Der (j,k)-te Eintrag dieser Produktmatrix ergibt sich zu:

$$\sum_{l=0}^{n-1} z^{jl} z^{-lk}/n = \frac{1}{n} \cdot \sum_{l=0}^{n-1} z^{l(j-k)} = \begin{cases} 1, & j = k \\ 0, & j \neq k \end{cases}$$

Das heißt, das Ergebnis ist die Einheitsmatrix – was zu beweisen war.

7.4 Euklidscher Algorithmus

Dieser Algorithmus dient der Bestimmung des *größten gemeinsamen Teilers* zweier ganzer Zahlen und geht auf Euklid (ca. 300 v.Chr.) zurück.

Algorithmus 7.3 `euklid`(a,b)

```
1: IF b = 0 THEN
2:   return a
3: ELSE
4:   return euklid(b, a mod b)
```

Beispiel:

$$\begin{aligned} \texttt{euklid}(21,30) &= \texttt{euklid}(30,21) \\ &= \texttt{euklid}(21,9) \\ &= \texttt{euklid}(9,3) \\ &= \texttt{euklid}(3,0) \\ &= 3 \end{aligned}$$

Die Korrektheit des Euklidschen Algorithmus beruht auf folgender Beobachtung:

$$ggt(a,b) = ggt(b, a \bmod b)$$

Beweis:
Wir zeigen

$$\{t \mid t \text{ teilt } a \text{ und } t \text{ teilt } b\} = \{s \mid s \text{ teilt } b \text{ und } s \text{ teilt } (a \bmod b)\}$$

woraus sich die Behauptung ergibt (denn wenn diese beiden endlichen Mengen gleich sind, müssen auch ihre größten Elemente gleich sein).

Im Folgenden reden wir immer über ganze Zahlen.

($\subseteq$) Sei $a = xt$ und $b = yt$. Dann gilt für die Zahl $r = (a \bmod b)$ die Darstellung $a = zb + r$, wobei $r < b$. Einsetzen ergibt: $xt = zyt + r$ bzw. $r = (x - yz)t$. Also ist t auch ein Teiler von r.

($\supseteq$) Sei $b = xs$ und $r = ys$, wobei $r = (a \bmod b)$. Für eine Zahl z gilt $a = zb + r$ mit $r < b$. Einsetzen ergibt: $a = zxs + ys = (zx + y)s$. Also ist s ein Teiler von a. □

Seien (a_i, b_i) die Parameter von `euklid` beim i-ten rekursiven Aufruf und seien (a_0, b_0) die Eingabewerte. Wir betrachten eine Folge von `euklid`-Aufrufen:

$$(a_0, b_0) \mapsto (a_1, b_1) \mapsto \cdots \mapsto (a_k, b_k)$$

Hierbei ist $b_k = 0$, $a_k = ggt(a_0, b_0)$. Ferner gilt $a_{i+1} = b_i$ und $b_{i+1} = (a_i \bmod b_i)$ für $i = 0, \ldots, k - 1$.

Wir behaupten, dass für $i = 0, \ldots, k - 2$ gilt: $b_i > 2b_{i+2}$.

Beweis:
Es gilt $b_i = a_{i+1} = xb_{i+1} + b_{i+2}$ für eine Zahl $x \geq 1$. Außerdem gilt $b_{i+2} < b_{i+1}$. Einsetzen von $x = 1$ ergibt die Abschätzung $b_i \geq b_{i+1} + b_{i+2} > 2b_{i+2}$, was zu zeigen war. □

Somit ergibt sich:

$$b_0 > 2b_2 > 4b_4 > 8b_6 > \cdots > 2^{(k-1)/2} b_{k-1} \geq 2^{(k-1)/2}$$

sofern k ungerade ist (den Fall, dass k gerade ist, kann man ähnlich behandeln). Hieraus folgt $b_0 > 2^{(k-1)/2}$ bzw. $k < 1 + 2 \log b_0$. Das heißt, die Anzahl der Iterationsschritte (rekursiven Aufrufe) von `euklid` ist $\mathcal{O}(\log b)$, wobei b der zweite Eingabeparameter ist.

In dieser Situation, bei der die Eingaben natürliche Zahlen sind, sollte mittels der Bit-Komplexität abgeschätzt werden. Wir nehmen an, die Eingabezahlen a, b besitzen höchstens n Bits in der Binärdarstellung. Also gilt $a, b < 2^n$. Die Bitkomplexität von `euklid` ist somit

$$\begin{aligned} &\text{(Anzahl Iterationen)} \cdot \text{(Bit-Komplexität für eine Iteration)} \\ &= \mathcal{O}(\log b) \cdot \mathcal{O}(n^2) = \mathcal{O}(n^3) \end{aligned}$$

Hierbei ist die Bit-Komplexität einer einzelnen Iteration im Wesentlichen durch den Aufwand für eine Multiplikation/Division gegeben, nämlich $\mathcal{O}(n^2)$.

Tatsächlich zeigt eine genauere Analyse, dass die Komplexität lediglich $\mathcal{O}(n^2)$ ist. Wenn man die Schulmethode zur Berechnung der Division, und damit der mod-Operation,

zugrundelegt, so sieht man, dass man zur Division einer n_1-Bit-Zahl durch eine n_2-Bit-Zahl $\mathcal{O}((n_1 - n_2) \cdot n_2)$ Bit-Operationen benötigt, denn der Divisor wird am Dividenden „entlanggeschoben“, welches $n_1 - n_2$ Positionen erfordert. Eine Iteration des Euklidschen Algorithmus, die vom Zahlenpaar a, b zum Zahlenpaar b, c mit $c = a \bmod b$ übergeht, erfordert somit

$$\mathcal{O}((l_a - l_b) \cdot l_b) \;=\; \mathcal{O}(l_a l_b - l_b^2) \;=\; \mathcal{O}(l_a l_b - l_b l_c)$$

Bit-Operationen, wobei l_x die Länge der Binärdarstellung der Zahl x ist. Somit erfordert eine Euklid-Berechnung, die die Zahlen $a_1, a_2, \ldots, a_m$ durchläuft, den Aufwand

$$\mathcal{O}(l_{a_1} l_{a_2} - l_{a_2} l_{a_3} + l_{a_2} l_{a_3} - l_{a_3} l_{a_4} + \cdots + l_{a_{m-2}} l_{a_{m-1}} - l_{a_{m-1}} l_{a_m}) \;=\; \mathcal{O}(l_{a_1} l_{a_2}) = \mathcal{O}(n^2)$$

7.5 Berechnen der modularen Exponentiation

Betrachten wir die Bit-Komplexität der verschiedenen Rechenoperationen modulo einer Zahl n. Hierbei seien a, b, n jeweils Zahlen mit höchstens m Bits. Die Addition von a und b (modulo n) kann man mit Komplexität $\mathcal{O}(m)$ ausführen, indem man zunächst a und b wie gewöhnlich bitweise addiert und dann ggfs. n subtrahiert. Für die Multiplikation benötigen wir $\mathcal{O}(m^2)$. Für das Bestimmen von Inversen mittels `extendedEuklid` haben wir die Komplexität $\mathcal{O}(m^3)$.

Aber wie sieht es mit der Berechnung der modularen Exponentiation, also $a^b \pmod n$ aus? Es ist nicht effizient, diese Funktion auf die Weise $\underbrace{a \cdot a \cdots a}_{b\text{-mal}} \pmod n$ zu berechnen. Dies ergäbe die Bit-Komplexität $\mathcal{O}(m^2 2^m)$.

Es geht wesentlich besser, indem man die Zahl b zunächst in ihre Binärdarstellung entwickelt: $b = (b_k b_{k-1} \ldots b_1 b_0)_2$. Anhand der Binärdarstellung kann man durch fortgesetztes Quadrieren (und Multiplikation mit a, falls $b_i = 1$) die Potenz berechnen. Der folgende Algorithmus führt dies aus.

Algorithmus 7.4 `modExp`(a, b, n)

```
1: //Berechnet a^b (mod n)
2: //Die Binärdarstellung von b sei b_k b_{k-1} ... b_1 b_0
3: d = 1
4: FOR i = k DOWNTO 0 DO
5:     d = (d · d) mod n
6:     IF b_i = 1 THEN
7:         d = (d · a) mod n
8: return d
```

Zur Begründung der Korrektheit dieser Prozedur kann man wie folgt argumentieren. Die ersten Bits (bis auf das letzte) der Binärdarstellung von b repräsentieren eine gewisse Zahl c; das letzte Bit von b sei $x \in \{0, 1\}$. Dann ist $b = 2c + x$. Angenommen, $d = a^c$. Dann gilt

$$a^b = a^{2c+x} = d^2 a^x = \begin{cases} d^2, & x = 0, \\ d^2 a, & x = 1. \end{cases}$$

Hiermit wird die Berechnung, die pro Schleifendurchlauf getätigt wird, begründet.

Da die Anzahl der Schleifendurchläufe gleich der Anzahl der Bits von b ist, und da pro Schleifendurchlauf höchstens 2 Multiplikationen nötig sind, ist die Bit-Komplexität von `modExp` $\mathcal{O}(m^3)$.

Man kann die Berechnung der Binärdarstellung der Zahl b, die in der obigen Darstellung separat zu erfolgen hat, auch sehr leicht in die Prozedur mit integrieren. Das Ergebnis sieht dann wie folgt aus:

Algorithmus 7.5 `fastExp`(a, b, n)

```
1: IF b = 1 THEN
2:    return a
3: ELSE IF odd(b) THEN
4:    return a · fastExp(a, b − 1, n) mod n
5: ELSE
6:    return square(fastexp(a, b div 2)) mod n
```

7.6 Primzahltesten

Wir erinnern uns, dass

$$\mathbb{Z}_n^* = \{a \in \{1, \ldots, n-1\} \mid ggt(a, n) = 1\}$$

Die *Euler-Funktion* ist definiert durch $\varphi(n) = |\mathbb{Z}_n^*|$. Es ist klar, dass immer $1 \leq \varphi(n) \leq n-1$ gilt, wobei $\varphi(n) = n-1$ genau dann gilt, wenn n eine Primzahl ist.

SATZ (Euler):
Für alle $n \geq 2$ und alle $a \in \mathbb{Z}_n^*$ gilt: $a^{\varphi(n)} \equiv 1$ (mod n).

Beweis:
Sei $\mathbb{Z}_n^* = \{z_1, z_2, \ldots, z_{\varphi(n)}\}$. Im Folgenden seien alle Multiplikationen und Gleichungen modulo n zu verstehen. Betrachte nun die Menge

$$a\mathbb{Z}_n^* = \{az_1, az_2, \ldots, az_{\varphi(n)}\}$$

Alle Elemente in dieser Menge sind paarweise verschieden (denn $z_i \neq z_j \Rightarrow az_i \neq az_j$). Daher gilt $\mathbb{Z}_n^* = a\mathbb{Z}_n^*$. Deshalb können wir folgern

$$\prod_{i=1}^{\varphi(n)} z_i = \prod_{i=1}^{\varphi(n)} (az_i) = a^{\varphi(n)} \cdot \prod_{i=1}^{\varphi(n)} z_i$$

da auf beiden Seiten des ersten Gleichheitszeichens über dieselbe Menge von Zahlen aufmultipliziert wird. Indem man beide Seiten mit $\left(\prod_{i=1}^{\varphi(n)} z_i\right)^{-1}$ multipliziert, folgt $1 = a^{\varphi(n)}$. □

Aus dem Satz von Euler ergibt sich durch spezielle Wahl von n als Primzahl der folgende Satz.

SATZ (Fermat):
Für alle Primzahlen n *und* $a \in \{1, \ldots, n-1\}$ *gilt* $a^{n-1} \equiv 1$ *(mod* n*).*

Die Umkehrung des Satzes von Fermat gilt leider nicht; aber doch „fast“, denn es gibt nur sehr wenige, rar vorkommende Gegenbeispiele. Das Folgende ist ein „fast richtiger“, effizienter Primzahltest:

Algorithmus 7.6 `pseudoPrim`(n)

```
1: IF modExp(2, n − 1, n) = 1 THEN
2:    return „wahrscheinlich Primzahl“
3: ELSE
4:    return „keine Primzahl“
```

Folgende „Fehlerquoten“ sind bekannt: Sei n eine Zufallszahl mit 50 (bzw. 100) Dezimalziffern. Dann ist die Wahrscheinlichkeit, dass n durch obigen Algorithmus fälschlicherweise als Primzahl deklariert wird, nicht größer als 10^{-6} (bzw. 10^{-13}). (Das erste Gegenbeispiel ist $341 = 11 \cdot 31$).

Man könnte die Fehlerquote noch weiter verkleinern, indem man nicht nur auf $2^{n-1} \equiv 1 \pmod{n}$, sondern zusätzlich auch auf $3^{n-1} \equiv 1 \pmod{n}$ (und evtl. weitere Basiszahlen) testet. Es lässt sich allerdings zeigen, dass es für jede feste Basiszahl a unendlich viele Zahlen n gibt, die den Fermat-Test $a^{n-1} \equiv 1 \pmod{n}$ bestehen und keine Primzahlen sind. Darüber hinaus gibt es Zahlen n, die keine Primzahlen sind, und die den Fermat-Test *für alle* $a \in \mathbb{Z}_n^*$ bestehen (sogenannte *Carmichael-Zahlen*). Die kleinsten Carmichael-Zahlen sind $561 = 3 \cdot 11 \cdot 17$ und $1729 = 7 \cdot 13 \cdot 19$.

Der nachfolgende Miller-Rabin-Primzahltest erweitert und verfeinert die obigen Ideen wie folgt: Die Basiszahl a für den Fermat-Test $a^{n-1} \stackrel{?}{\equiv} 1 \pmod{n}$ wird zufällig gewählt. sollte n diesen Test bestehen, so muss n noch keine Primzahl sein. Insbesondere könnte n eine Carmichael-Zahl sein. Wenn man nun $a^{\frac{n-1}{2}} \bmod n$ betrachtet (n ist eine gerade Zahl), so ist dies eine modulare Quadratwurzel der Zahl 1, denn

$$\left(a^{\frac{n-1}{2}}\right)^2 = a^{n-1} \equiv 1 \pmod{n}$$

Sofern n eine Primzahl ist, so bestitzt die 1 genau die zwei Quadratwurzeln 1 und -1 (entspricht $n-1$). Sollte $a^{\frac{n-1}{2}}$ einen Wert ungleich 1 und ungleich $n-1$ haben, so ist n mit Sicherheit keine Primzahl. Sollte $a^{\frac{n-1}{2}}$ gleich 1 sein, so könnte man auch noch $a^{\frac{n-1}{4}}$ berechnen (sofern $\frac{n-1}{2}$ eine gerade Zahl ist).

Die Berechnung dieser fraglichen Zahlen geschieht im Rahmen der modularen Exponentiationsberechnung von $a^{n-1} \bmod n$ sowieso, also kann man diesen zusätzlichen „Quadratwurzeltest“ in die modulare Exponentiation integrieren.

Algorithmus 7.7 `test`(a, n)

```
1: //Sei b_k ... b_0 die Binärdarstellung von n − 1
2: d = 1
3: FOR i = k DOWNTO 0 DO
4:    x = d
5:    d = (d · d) mod n
6:    IF d = 1 ∧ x ≠ 1 ∧ x ≠ n − 1 THEN
7:       return TRUE //bedeutet: n ist keine Primzahl
8:    IF b_i = 1 THEN
9:       d = (d · a) mod n
10: return d ≠ 1
```

Aufgerufen wird dieser Test vom nachfolgenden eigentlichen Programm. Hierbei ist s ein „Sicherheitsparameter“, über den die Fehlerwahrscheinlichkeit eingestellt werden kann.

Algorithmus 7.8 `millerRabin`(n, s)

```
1: FOR i = 1 TO s DO
2:    random a ∈ {1, ..., n − 1}
3:    IF test(a, n) THEN
4:       return „keine Primzahl“
5: return „Primzahl“
```

Man kann zeigen (hier ohne Beweis), dass die Wahrscheinlichkeit, fälschlicherweise „Primzahl“ auszugeben, pro Aufruf von `test`(a, n) höchstens 1/4 ist. Die Prozedur `test` wird insgesamt s mal aufgerufen. Daher ist die Wahrscheinlichkeit, jedes Mal einen Fehler zu machen, höchstens $(1/4)^s$. Um also die Fehlerwahrscheinlichkeit von zum Beispiel 10^{-10} zu erreichen, muss man $s = 17$ wählen.

Viele kryptographischen Anwendungen benötigen zufällige Primzahlen mit etwa 100 Dezimalstellen. Diese kann man wie folgt erhalten:

Algorithmus 7.9 `getPrime`()

```
1: REPEAT
2:    Wähle eine 100-stellige Zufallszahl n
3: UNTIL n ist Primzahl (gemäß millerRabin)
```

Nun könnte es sein, dass man viel zu viele Schleifendurchläufe machen muss, bis man eine Primzahl gefunden hat. Dass dies aber nicht so ist, garantiert der Primzahlsatz: für die Anzahl der Primzahlen bis zur Zahl n gilt $\pi(n) \sim n/\ln n$. Daher ist die Wahrscheinlichkeit im Intervall $\{1, \ldots, n\}$ bei zufälliger Auswahl eine Primzahl zu finden, etwa $1/\ln n$. Das heißt, im Durchschnitt benötigt man $\ln n$ viele Versuche, bis man eine Primzahl gefunden hat. Beispielweise gilt $\ln 10^{100} \approx 230$. (Übrigens braucht man von vornherein nur ungerade Zahlen zu testen und reduziert damit den Aufwand auf die Hälfte).

Eine weitere Bemerkung: Wir haben also gesehen, dass es effiziente Primzahltests gibt (mit dem Abstrich, dass diese probabilistisch sind und eine gewisse Fehlerquote in sich tragen).Wenn der Primzahltest aber „keine Primzahl“ ausgibt, so handelt es sich um eine Zahl, die sich in nicht-triviale Faktoren zerlegen lässt. Wir sind aber weit davon entfernt, solche Faktoren in effizienter Weise zu erhalten. Nehmen wir an, n sei das Produkt zweier 100-stelliger Primzahlen p und q. Die Aufgabe, aus einer solchen gegebenen Zahl $n = pq$ die Faktoren p und q rückzugewinnen, ist auch mit heutigen Computern eine „praktisch unlösbare“ Aufgabe. Eine solche Berechnung würde Jahrtausende dauern.

Viele kryptographische Anwendungen (z.B. das RSA-Verfahren) bedienen sich genau dieser Diskrepanz: Primzahl-Testen geht einfach; Primfaktoren zu finden, ist schwierig.

7.7 Faktorisierung: Pollards ρ-Algorithmus

Zunächst bemerken wir, dass es zum Faktorisieren einer Zahl n genügt, zunächst zwei nicht-triviale Faktoren $n = p \cdot q$ zu bestimmen. Diese Faktoren können daraufhin mit einem Primzahltest getestet werden, ob sie Primzahlen sind, und wenn nicht, so kann dieser Algorithmus zur Bestimmung zweier nicht-trivialer Faktoren weiter rekursiv angewendet werden.

Der naive Ansatz zum Faktorisieren einer Zahl n besteht darin, alle potenziellen Teiler durchzuprobieren. Hierbei genügt es, nur die Zahlen bis $\sqrt{n}$ zu testen, denn wenn n einen nicht-trivialen Teiler besitzt (also keine Primzahl ist), so gibt es einen Teiler $\leq \sqrt{n}$. (Ausserdem braucht man außer der 2 nur ungerade Probeteiler zu durchlaufen). Dieses Verfahren hat die Komplexität $\mathcal{O}(\sqrt{n})$. Dies ist im Sinne der Bit-Komplexität $\mathcal{O}(\sqrt{2^m}) = \mathcal{O}(2^{m/2})$, also exponentiell (wenn m die Länge der Binärdarstellung von n ist). Es ist, wie oben gesagt, kein effizienter Algorithmus für das Faktorisieren bekannt, wir wollen hier aber eine Verbesserung der naiven Methode besprechen, die die Komplexität $\mathcal{O}(2^{m/4})$ hat.

Für alle Faktorisierungsverfahren gilt, dass man gut daran tut, sich zunächst mit einem schnellen Primzahltest davon zu überzeugen, dass es sich bei der Eingabe n *nicht* um eine Primzahl handelt. Anderernfalls würde der Faktorisierungsalgorithmus, auf Eingabe n, unnötig lange rechnen, ohne je zu einem Ergebnis zu gelangen.

Der ρ-Algorithmus von Pollard beruht auf folgender Idee: Angenommen, wir haben eine zufällige Folge von Zahlen $x_0, x_1, x_2, \ldots$, wobei $x_i \in \{0, 1, \ldots, n-1\}$, zur Verfügung. Sei nun $n = pq$ eine Faktorisierung von n, wobei p eine Primzahl und $p \leq q$, also $p \leq \sqrt{n}$ ist. Mit gewisser Wahrscheinlichkeit wird es nach nicht allzulanger Zeit Indizes $i < j$ geben mit $x_i \equiv x_j \pmod{p}$ (obwohl wir dies nicht explizit testen können, denn p ist ja gerade gesucht). Aufgrund des Geburtstagsparadoxons (siehe Seite 45) tritt dieses Ereignis im Mittel nach $\mathcal{O}(p^{1/2}) = \mathcal{O}(n^{1/4})$ vielen Schritten ein. Sofern solche Indizes i, j vorliegen, gilt, dass $|x_i - x_j|$ ein Vielfaches von p ist. Mit hoher Wahrscheinlichkeit ist dann der größte gemeinsame Teiler von $|x_i - x_j|$ und n ein nicht-trivialer Teiler von n (nämlich p oder ein Vielfaches von p). Mit sehr geringer Wahrscheinlichkeit könnte jedoch $x_i \equiv x_j \pmod{n}$ gelten, in diesem Fall erhalten wir keinen Teiler von n (ausser n selbst).

Nehmen wir an, die Zufallszahlen $x_0, x_1, \ldots$ sind eigentlich nur *Pseudo*-Zufallszahlen. Das heißt, nur x_0 (der „seed“) ist eine echte Zufallszahl und die weiteren Zahlen

$x_1, x_2, \ldots$ werden aufgrund einer deterministischen Vorschrift $x_{i+1} := f(x_i) \bmod n$ berechnet. Sofern für i, j gilt $x_i \equiv x_j \pmod p$ und sofern die Funktion f ein Polynom ist (d.h. es kommt nur Addition und Multiplikation vor), so folgt für alle k, dass $x_{i+k} \equiv x_{j+k} \pmod p$. (Was wir hier auch benötigen ist die Tatsache, dass $(x \bmod n) \bmod p = x \bmod p$). Von einem bestimmten Zeitpunkt an befindet sich die Folge der x_i in einer Schleife, deren Zyklenlänge $j - i$ ist.

Skizze (für $i = 3$ und $j = 9$):

$$\begin{array}{ccccccccccc} x_0 & \to & x_1 & \to & x_2 & \to & x_3 = x_9 & \to & x_4 & \to & x_5 \\ & & & & & & \uparrow & & & & \downarrow \\ & & & & & & x_8 & \leftarrow & x_7 & \leftarrow & x_6 \end{array}$$

Aus der Form dieses Diagramms wird auch die Bezeichnung „ρ-Algorithmus“ offensichtlich.

Anstatt nun alle Indexpaare i, j durchzuprobieren, genügt es, nur die Kombinationen $x_1 - x_2$, $x_2 - x_4$, $x_3 - x_6$, $x_4 - x_8 \ldots$, allgemein $x_t - x_{2t}$ ($t = 1, 2, 3, \ldots$) zu generieren. Denn wenn $x_i \equiv x_j \pmod p$, $i < j$, gilt, so wird im Rahmen dieses Suchprozesses $x_t \equiv x_{2t} \pmod p$ gelten, wobei für t gelten muss, dass es ein Vielfaches von $j - i$ ist und dass $t \geq i$ gilt. Bei obigem Beispiel hätten wir mit der Kombination $x_6 - x_{12}$ Erfolg.

Beobachtung: Für $i < j$ gelte $x_i \equiv x_j \pmod p$. Dann gibt es ein $t \leq j$ mit $x_t \equiv x_{2t} \pmod p$.

Damit haben wir das Grundschema für den ρ-Algorithmus:

Algorithmus 7.10 $\rho(n)$

```
1: //Eingabe: eine Nicht-Primzahl n
2: random x ∈ [0..n − 1]
3: y = f(x) mod n
4: REPEAT
5:    IF x ≠ y THEN
6:       g = ggt(abs(x − y), n)
7:       IF g > 1 THEN
8:          output g
9:    x = f(x) mod n
10:   y = f(f(y) mod n) mod n
11: UNTIL Resettaste gedrückt
```

Eine gängige Wahl für die Funktion f ist $f(x) = x^2 + 1$ oder $f(x) = x^2 - 1$.

Da wir die echten Zufallszahlen nun durch Pseudozufallszahlen ersetzt haben, sind die obigen Wahrscheinlichkeitsbetrachtungen und die daraus abgeleitete $\mathcal{O}(n^{1/4})$-Schranke nicht exakt, sondern nur heuristisch zu verstehen, werden aber durch Experimente bestätigt. Insbesondere besteht keine Garantie, dass der ρ-Algorithmus auch bei einer Nicht-Primzahl als Eingabe (in angemessener Zeit) stoppt.

Beispiel: Wir geben die Zahl $n = 12349 = 53 \cdot 233$ in den Algorithmus ein. Der Startwert für x sei 1 und wir wählen $f(x) = x^2 + 1$. Dann ergeben sich die folgenden Werte, wobei

nach 8 Schleifendurchläufen der Faktor 53 gefunden wird:

x	1	3	10	101	10202	3433	4544	409
y	3	101	3433	409	1310	7564	9313	6928
g	1	1	1	1	1	1	1	53

Man verifiziert, dass für die letzten Werte von x und y (409 und 6928) tatsächlich gilt $x \equiv y \pmod{53}$.

Wir bemerken, dass eine lineare Funktion $f(x) = ax + b$ im allgemeinen nicht so günstig ist wie eine quadratische Funktion, da in diesem Fall die Abbildung $x \mapsto f(x) \bmod p$ auf $\{0, 1, \ldots, p-1\}$ *injektiv* ist. Das heißt, ein linearer Kongruenzgenerator erfüllt nicht die für die Komplexitätsanalyse des Verfahrens wichtige statistische Aussage, dass die mittlere Anzahl von Schritten bis zum Erreichen der „ρ-Schleife“ (und auch deren Länge) $\mathcal{O}(\sqrt{p})$ ist.

Kapitel 8

Optimierung von Baumstrukturen

In diesem Kapitel werden Algorithmen behandelt, denen zugrunde liegt, dass ein sehr großer Baum (i.a. mit exponentieller Größe) nach einer „Lösung“ durchmustert werden muss, oder dass aus den Informationen in den Baumknoten ein Wert berechnet werden muss. Hierbei ist der Baum allerdings nicht explizit als Eingabe für den Algorithmus gegeben (sonst würde das Thema in das Kapitel „Algorithmen auf Graphen“ passen), sondern die Baumstruktur entsteht aus der eigentlichen Eingabe erst dynamisch während des Algorithmenablaufs (im Rahmen der verwendeten Rekursion). Ziel dieser Algorithmen ist es, möglichst nicht den vollständigen Baum zu generieren bzw. zu durchlaufen, sondern Teile des Baumes „abzuschneiden“, die zur Evaluierung nicht benötigt werden. Dies ist meist dadurch möglich, dass eine geschickte Reihenfolge bei der Auswertung des Baumes gewählt wird.

8.1 Backtracking

Falls die Lösung eines Problems gesucht ist, welche sich aus n Komponenten zusammensetzt und es für jede der Komponenten evtl. mehrere Wahlmöglichkeiten gibt, dann kann man sukzessive über rekursive Aufrufe (bis zur Rekursionstiefe n) nach der Gesamtlösung suchen, wobei über die verschiedenen rekursiven Aufrufe eine „Teillösung“, beginnend mit der leeren Lösung, zusammengesetzt wird. Wenn sich herausstellt, dass eine Teillösung nicht weiter fortsetzbar ist, um eine Gesamtlösung zu erhalten, so wird in die aufrufende Prozedur(inkarnation) zurückgesetzt (backtracking) und damit die „Sackgasse“ verlassen und eine andere Fortsetzung versucht.

Das generelle Schema lässt sich wie folgt beschreiben:

Algorithmus 8.1 `backtrack(Teillösung)`

```
1: IF vollständige Lösung erreicht THEN
2:   Gib Lösung aus
3: ELSE
4:   FOR jede zulässige Erweiterung der Teillösung DO
5:     backtrack(Teillösung mit Erweiterung)
6: return
```

Der Aufruf im Hauptprogramm lautet dann `backtrack`(leere Lösung).

Beispiel:
Betrachten wir das NP-vollständige Problem KNF-SAT. Gegeben sei eine Boolesche Formel F in konjunktiver Normalform, also eine Und-Verknüpfung von sog. Klauseln, welche wiederum Oder-Verknüpfungen von Variablen oder negierten Variablen sind. Die Aufgabe besteht darin, festzustellen, ob die Formel F erfüllbar ist, also ob es eine Belegung der Variablen mit Wahrheitswerten gibt, so dass die ausgewertete Formel den Wahrheitswert 1 erhält. Die vorkommenden Variablen seien $x_1, \ldots, x_n$. Indem wir diese Variablen sukzessive mit Wahrheitswerten belegen, erhalten wir partielle Belegungen. Solche partiellen Belegungen geben wir durch Strings $a_1 a_2 \ldots a_k$, $k \leq n$, $a_i \in \{0, 1\}$, an. An der Baumstruktur, die sich hierdurch implizit ergibt, wird der Backtracking-Mechanismus vollzogen. Eine „Sackgasse“, also ein vorzeitiges Beenden eines Belegungs-Versuchs, ergibt sich dann, wenn die bis dahin erzeugte partielle Belegung bereits eine der Klauseln mit dem Wert 0 belegt. In diesem Fall kann keine Erweiterung dieser partiellen Belegung mehr erfüllend sein, und diese brauchen daher nicht betrachtet zu werden. Der Algorithmus hierzu sieht wie folgt aus:

Algorithmus 8.2 `suche(`a`: partielle Belegung)`

```
1: //Liefert TRUE, genau dann wenn sich die partielle Belegung a
   zu einer erfüllenden Belegung für F erweitern lässt
2: IF a belegt alle Variablen THEN
3:   return F(a)
4: ELSE IF eine der Klauseln in F wird durch a auf 0 gesetzt THEN
5:   return FALSE //Sackgasse
6: ELSE
7:   IF suche(a0) THEN
8:     return TRUE
9:   ELSE
10:    return suche(a1)
```

Aufgerufen wird die Prozedur mittels `suche`(ε), wobei ε die leere Belegung ist.

Obwohl die Möglichkeit besteht, dass dieser Algorithmus eine erfüllende Belegung schnell findet, muss man im worst-case mit der Komplexität $\mathcal{O}(2^n)$ rechnen, da ein Prozeduraufruf von `suche`(a) zwei weitere Aufrufe, nämlich von `suche`$(a0)$ und `suche`$(a1)$ nach sich ziehen kann.

8.2 Branch-and-Bound

Die Methode Branch-and-Bound ist zur Lösung von Optimierungsproblemen angebracht, bei denen keine anderen effizienten Verfahren bekannt sind (z.B. bei NP-vollständigen Optimierungsproblemen. Hierbei muss der Lösungsraum (implizit) so strukturiert sein, dass er eine Baumstruktur darstellt. Hierbei entspricht die leere Lösung, die noch zu jeder beliebigen Lösung erweitert werden kann, der Wurzel. Dieser Baum, oder ein Teil davon, wird nach einem bestimmten Prinzip erzeugt, das an den Dijkstra-Algorithmus aus Abschnitt 5.4 erinnert (nur ist hier der Baum nicht explizit als Eingabe gegeben).

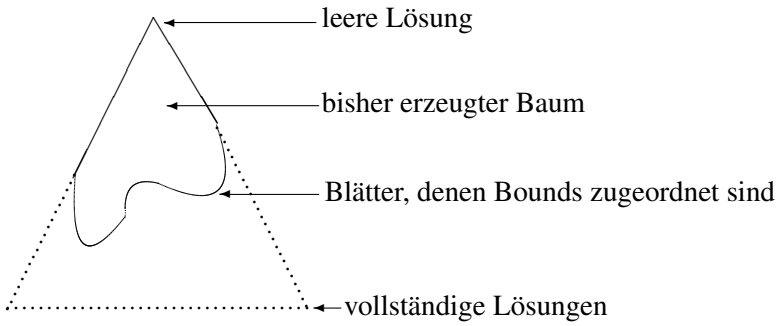
Beispiel:
Beim Traveling Salesman Problem (vgl. Abschnitt 4.3) kann man wie folgt eine Baumstruktur erzeugen. Gegeben sei eine Entfernungsmatrix $M[1..n, 1..n]$, die die Entfernungen der Städte $1, \ldots, n$ untereinander angibt. Für das Folgende ist es am besten, Wegeverbindungen, die nicht vorhanden sind oder nicht mehr gewählt werden können, mit dem Eintrag ∞ zu versehen. Insbesondere gehen wir bei der Ausgangsmatrix davon aus, dass $M[i, i] = \infty$ für alle $i = 1, \ldots, n$ gilt. Die gegebene Entfernungsmatrix M entspricht der leeren Lösung und damit der Baumwurzel; d.h. alle potenziellen Lösungen sind durch Erweiterung der leeren Lösung noch möglich. Sobald in der Entfernungsmatrix weitere Einträge (i, j) auf ∞ gesetzt werden, bleiben nur noch Lösungsmöglichkeiten übrig, die die Kante (i, j) *nicht* enthalten. Eine eindeutige und endgültige Lösung entsteht, wenn die Einträge der Entfernungsmatrix, die nicht auf ∞ gesetzt wurden, eine *Permutationsmatrix* bilden. Dann hat in jeder Zeile i genau ein Element (i, j_i) einen endlichen Wert und die Zahlen $(j_1, j_2, \ldots, j_n)$ bilden eine Permutation der Zahlen von 1 bis n (nämlich die Rundreise). Solche Permutationsmatrizen entstehen an den Blättern des Branch-and-Bound Baumes.

Den Verzweigungsschritt (Branch) organisieren wir solcherart, dass wir zunächst einen Eintrag (i, j) der Matrix auswählen, der nicht unendlich ist. Die beiden Nachfolger der Matrix M sind die beiden neuen Entfernungsmatrizen M' und M'', wobei in M' bzw. M'' einige Einträge solcherart auf unendlich gesetzt werden, dass M' diejenigen Rundreisen symbolisiert, die die Kante (i, j) benutzen, während für M'' diejenigen Rundreisen relevant sind, die die Kante (i, j) *nicht* benutzen. Wir erreichen dies dadurch, dass wir $M''[i, j] = \infty$ setzen. Was M' betrifft, so setzen wir $M'[i, k] = \infty$ für $k \neq j$ und $M'[l, j] = \infty$ für $l \neq i$. Ferner setzen wir $M'[j, i] = \infty$. Alle diese Kantenauswahlen können ausgeschlossen werden, wenn die Rundreise über (i, j) verläuft.

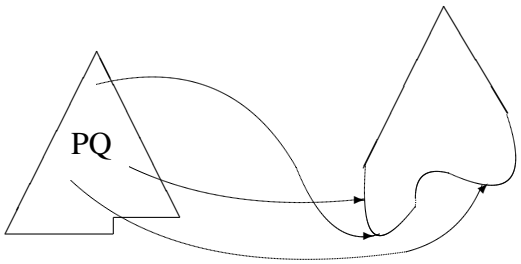
Sobald im Branch-and-Bound Baum ein neuer Knoten erzeugt wurde, wird für diesen eine gewisse untere Schranke b (der *Bound*) berechnet. Diese Schranke besagt, dass keine endgültige Lösung, die sich unterhalb des betreffenden Knotens befindet, einen kleineren (also besseren) Wert als b haben kann. Das bedeutet, dass alle Lösungen, die Erweiterungen der betreffenden partiellen Lösung darstellen, einen Wert haben müssen, der $\geq$ dem berechneten *Bound* für die partielle Lösung ist. (Wir gehen hier von einem Minimierungsproblem aus; bei einem Maximierungsproblem müsste der *Bound* eine *obere* Schranke sein). Wie ein solcher *Bound* berechnet werden kann, ist problem-spezifisch und wird weiter unten am Beispiel des TSP erläutert.

Der berechnete *Bound* liefert ein Präferenzkriterium. Im nächsten Schritt, dem Branch-Schritt, werden die *Bounds* aller bisherigen Baumblätter miteinander verglichen, und dasjenige Blatt mit dem kleinsten *Bound* wird weiter expandiert; das heißt, es werden seine Nachfolger generiert, die zugehörige partielle Lösung wird also um eine Komponente erweitert. Danach werden die *Bounds* der Nachfolger bestimmt, und so weiter. Um dasjenige Blatt mit dem kleinsten *Bound* effizient zu bestimmen, empfiehlt es sich, die Information über die *Bounds* der aktuellen Baumblätter in einer Priority Queue zu organisieren (vgl. Abschnitt 5.6).

Das folgende Bild skizziert die Situation:



Im folgenden Bild ist skizziert, wie eine Priority Queue (hier implementiert als Heap) parallel zum ständig wachsenden Branch-and-Bound Baum über die *Bounds* an den Blättern Buch führt, um möglichst schnell dasjenige Blatt mit dem kleinsten *Bound* auffinden zu können. Von jedem Heap-Knoten führt ein Zeiger auf ein Blatt des Branch-and-Bound Baumes. Die Heap-Wurzel zeigt auf das Blatt mit dem kleinsten *Bound.*



Wenn bei diesem Verfahren ein Ast so weit entwickelt wurde, dass das entsprechende Blatt eine vollständige Lösung darstellt und dessen Wert im Vergleich zu allen anderen *Bounds* minimal ist, dann ist garantiert, dass diese Lösung optimal ist, denn alle anderen momentanen Baumblätter können höchstens noch zu endgültigen Lösungen erweitert werden mit einem schlechteren oder demselben Wert.

Die Idee des Verfahrens ist also, dass viele der Verzweigungen nicht weiter betrachtet werden müssen, da sie einen zu großen *Bound* erhalten haben. Daher braucht der Baum i.a. nicht vollständig entwickelt zu werden und die Komplexität bleibt im Erträglichen. (Im worst-case hat die Methode aber i.a. exponentielle Komplexität).

Man macht sich leicht klar, dass die Güte der *Bounds* für die Effizienz des Verfahrens von entscheidender Bedeutung ist. Nehmen wir im Extremfall mal an, dass wir die Möglichkeit hätten, die *Bounds* exakt zu berechnen. Das soll heißen, dass der *Bound* einer

partiellen Lösung tatsächlich immer mit dem Minimum aller derjenigen Lösungswerte übereinstimmt, die sich aus der partiellen Lösung entwickeln lassen. Insbesondere hätten wir dann bereits an der Wurzel, die der leeren Lösung entspricht, die Möglichkeit exakt zu bestimmen, welche Güte die optimale Lösung hat. In diesem Fall würde der Branch-and-Bound Algorithmus zielstrebig einen einzigen Pfad aus der leeren Lösung heraus entwickeln, der direkt auf die optimale Lösung zusteuert.

Oftmals erhält man gute *Bounds* für eine Problemstellung durch eine *Relaxation*, das heißt, die zugrundeliegende Aufgabenstellung wird „aufgeweicht“ (es werden zum Beispiel Bedingungen, die eigentlich zu beachten sind, weggelassen etc.), so dass effiziente Lösungsmöglichkeiten für die so veränderte Aufgabenstellung bestehen. Dies ist dann zwar nicht mehr die ursprüngliche Aufgabenstellung, da der Lösungsraum vergrößert wurde, aber die Lösungsgüte der Relaxation stellt eine Schranke für die eigentliche, gesuchte Lösung dar, die im Rahmen von Branch-and-Bound als *Bound* verwendet werden kann. Dies liegt daran, dass das eigentliche zu lösende Problem im vergrößerten Lösungsraum des relaxierten Problems mit enthalten ist. (Weiteres zum Begriff Relaxation findet man in Abschnitt 10.1).

Wir betrachten wieder das Beispiel des Traveling Salesman Problems. Wie bestimmen wir den *Bound*, also eine (möglichst gute) untere Schranke für die Güte der unterhalb von M erreichbaren Lösung? Hierzu beobachten wir folgendes. Seien $a_1, a_2, \ldots, a_n$ die Werte einer festgehaltenen Zeile (oder Spalte) von M. Sei $m = \min\{a_1, a_2, \ldots, a_n\}$. Indem wir diese Zeile (bzw. Spalte) von M modifizieren zu $a_1 - m, a_2 - m, \ldots, a_n - m$ erhalten wir eine neue Entfernungsmatrix $\hat{M}$ mit der Eigenschaft, dass für jede Lösung π von M mit Wert c die entsprechende Lösung π für $\hat{M}$ den Wert $c - m$ hat (und umgekehrt). Das heißt, wenn π eine optimale Lösung für M ist mit minimalem (Entfernungs-) Wert, so ist π auch minimal für $\hat{M}$ mit einem um m reduzierten Wert. Die Optimierungsprobleme M und $\hat{M}$ sind also – bis auf die Werteverschiebung um m – äquivalent.

Den einer Matrix M zugeordneten *Bound* berechnen wir algorithmisch wie folgt: Wir reduzieren jede Zeile von M und erhalten hierbei die Reduktionswerte $m(1), \ldots, m(n)$. Dann reduzieren wir noch jede Spalte und erhalten die Werte $m(n+1), \ldots, m(2n)$. Der *Bound* für die Matrix M ist dann $\sum_{i=1}^{2n} m(i)$.

Der folgende Algorithmus berechnet den *Bound* von M.

Algorithmus 8.3 `bound`(M)

```
 1: b = 0
 2: FOR i = 1 TO n DO
 3:    min = ∞
 4:    FOR j = 1 TO n DO
 5:       IF M[i,j] < min THEN
 6:          min = M[i,j]
 7:    b = b + min
 8:    FOR j = 1 TO n DO
 9:       M[i,j] = M[i,j] − min
10: FOR j = 1 TO n DO
11:    min = ∞
12:    FOR i = 1 TO n DO
13:       IF M[i,j] < min THEN
14:          min = M[i,j]
15:    b = b + min
16:    FOR i = 1 TO n DO
17:       M[i,j] = M[i,j] − min
18: return b
```

Beispiel: Die folgende Matrix M

$$\begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix}$$

wird wie folgt zunächst zeilen-reduziert:

$$\begin{bmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{bmatrix}$$

was den Wert $10 + 5 + 6 + 8 = 29$ ergibt. Die anschließende Reduktion nach den Spalten liefert

$$\begin{bmatrix} \infty & 0 & 4 & 5 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{bmatrix}$$

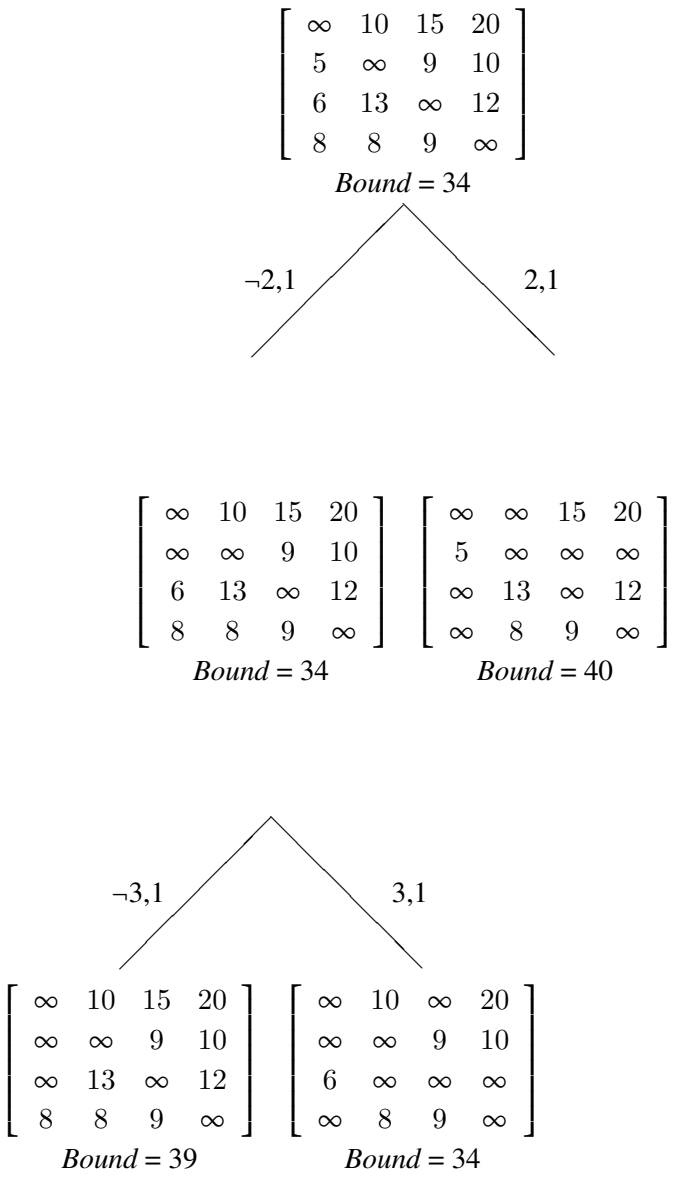
was einen weiteren Wertgewinn von 5 ergibt. Dies macht zusammen 34. Also ist *Bound*(M)=34.

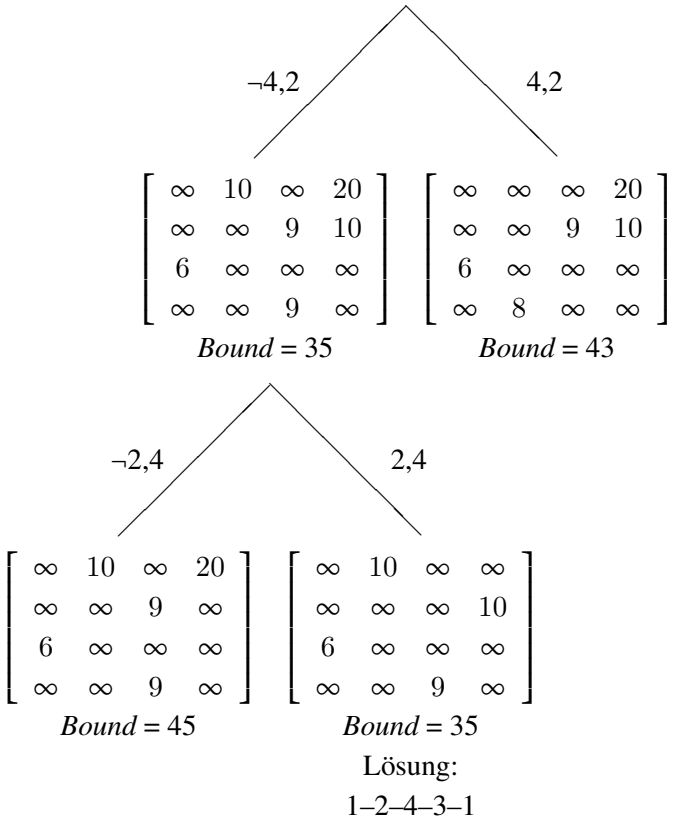
Dies ist eine untere Schranke für die zu erwartende Rundreiselänge in M, denn eine optimale Rundreise in M entspricht einer optimalen Rundreise in der reduzierten Matrix.

Letztere hat mindestens den Wert 0. Also hat die kürzeste Rundreise in M mindestens den Wert $\sum_{i=1}^{2n} m(i)$.

Ganz allgemein kann man die typische Vorgehensweise zur Bestimmung von aussagekräftigen *Bounds* so beschreiben: Man betrachtet eine Relaxation des eigentlichen Problems und erhält damit – mit vergleichsweise geringem Komplexitätsaufwand – eine Lösung der Relaxation, deren Wert als untere Schranke für die Lösung des eigentlichen Problems dient.

Das folgende *Beispiel* zeigt den Ablauf des Branch-and-Bound Algorithmus (der von Little, Murty, Sweeney, Karel (1963) vorgeschlagen wurde).





Die ermittelte Lösung mit dem *Bound* 35 (was der tatsächlichen Länge der Rundreise entspricht) ist nun garantiertermaßen optimal, da es im entwickelten Baum kein Blatt gibt, das einen geringeren *Bound* hat und damit noch zu einer besseren Lösung führen könnte. Dies ist das Abbruchkriterium für den Algorithmus: wenn eine vollständige Lösung entwickelt wurde, dessen Wert mindestens so gut ist wie jeder andere *Bound* an den Blättern des insoweit erzeugten Baumes.

Zusammenfassend sind die wesentlichen Bestandteile des Branch-and-Bound Ansatzes die Folgenden: Erstens muss eine Baumstruktur über den Lösungsraum gelegt werden; auf diese Weise werden die Branch-Schritte definiert. Zweitens muss eine Methode gefunden werden, wie man möglichst gute *Bounds* berechnen kann, um so den zu entwickelnden Baum nicht allzusehr „auswuchern“ zu lassen, sondern möglichst zielstrebig auf die optimale Lösung zuzusteuern.

Manchmal wird nur eine (optimistische) *Schätzung* für die Güte der zu erwartenden optimalen Lösung verwendet – also nicht notwendigerweise eine untere Schranke; gelegentlich sind solche Schätzungen viel einfacher zu erhalten als eine garantierte untere Schranke. Wenn man eine solche Schätzfunktion für den *Bound* in dem Verfahren einsetzt, so muss nicht mehr garantiertermaßen die optimale Lösung gefunden werden. In diesem Fall wird die Branch-and-Bound Methode zu einer *approximativen* Methode. Aber möglicherweise wird auf diese Weise in recht effizienter Weise eine nahezu optimale Lösung gefunden. Diese Thematik wird noch in Kapitel 10 vertieft.

Branch-and-Bound ist natürlich in symmetrischer Weise auch auf *Maximierungsprobleme* anwendbar (wobei mit dem *Bound* dann natürlich eine *obere* Schranke gemeint ist).

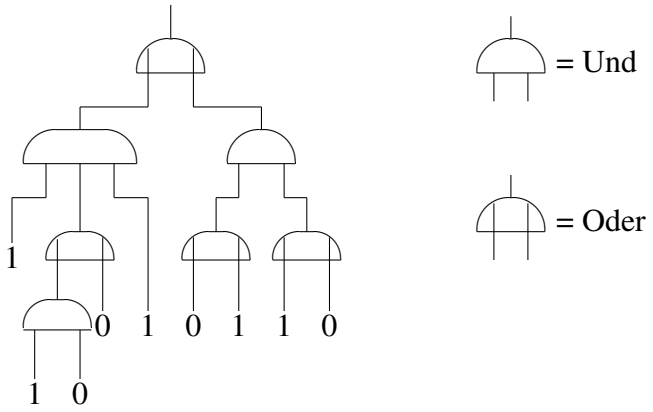
8.3 Und-Oder-Bäume

Manche Problemstellungen führen in natürlicher Weise zu sog. *Und-Oder-Bäumen*. Wenn man die gegebene Aufgabenstellung so in Teilprobleme zerlegen kann, dass aus der Lösung *aller* Teilprobleme die Gesamtlösung folgt (wie bei divide-and-conquer), so entspricht dies einer Und-Verzweigung.

Wenn dagegen das Problem so in Teilprobleme zerlegt werden kann, dass es schon genügt, *eines* des Teilprobleme zu lösen, so entspricht dies einer Oder-Verzweigung.

Das ursprüngliche Problem kann solcherart in verschiedene Teilprobleme zerlegt werden und diese weiter zerlegt werden etc., so dass ein Baum von alternierenden Und- und Oder-Verzweigungen entsteht. An den Blättern dieses Und-Oder-Baumes sind Teilprobleme eingetragen, die nicht weiter zerlegt werden können und elementar gelöst werden müssen (sofern sie lösbar sind).

Die abstrakte, hinter dieser Zerlegung stehende Aufgabe besteht darin, bei einem solcherart gegebenen Und-Oder-Baum, samt einer Bewertung der Blätter mit Wahrheitswerten (1=lösbar, 0=nicht lösbar), festzustellen, ob eine Lösung des Gesamtproblems möglich ist. Das folgende Bild zeigt anschaulich, dass es sich letztlich um das Evaluieren einer Booleschen Schaltung handelt, die aus Und- und Oder-Gattern besteht.



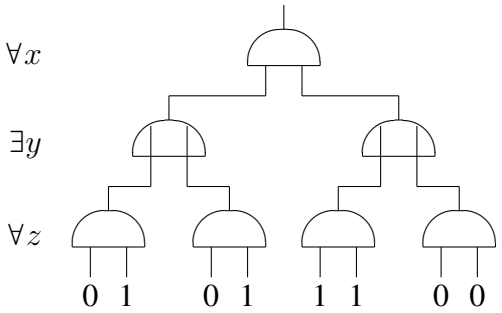
Das Besondere ist nur, dass der Und-Oder-Baum nicht unbedingt explizit gegeben ist, sondern sich nur implizit aus der gegebenen Aufgabenstellung ergibt.

Beispiel:
Sogenannte *quantifizierte Boolesche Formeln*, kurz QBF, sind Boolesche Formeln mit Quantoren, wobei sich der Allquantor und der Existenzquantor auf Variablen beziehen, die die beiden Wahrheitswerte 0 und 1 annehmen können. Beispielsweise ist

$$F \;=\; \forall x\, \exists y\, \forall z\, \big((x \wedge \overline{y}) \vee (\overline{x} \wedge z)\big)$$

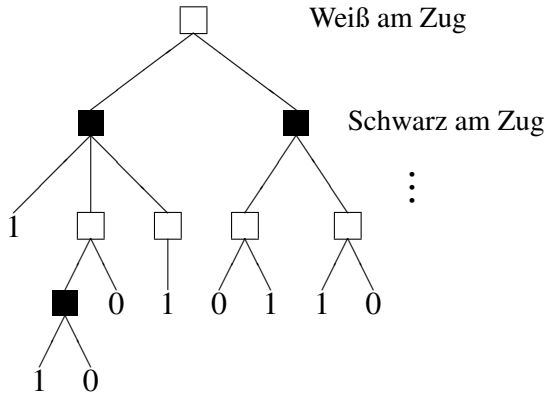
eine QBF. Da alle vorkommenden Variablen durch Quantoren gebunden sind, lässt sich diese Formel eindeutig auswerten und ergibt einen Wahrheitswert. Das algorithmische Problem, QBFs auszuwerten, ist NP-schwierig (genauer: es ist „PSPACE-vollständig“). Daher sind keine effizienten Algorithmen zur Auswertung von QBFs bekannt.

Das Auswerten einer QBF entspricht wieder dem Auswerten eines Und-Oder-Baums:



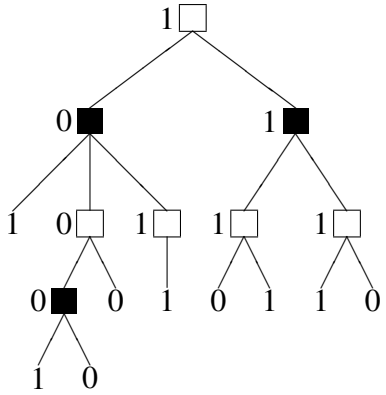
Die Wahrheitswerte an den Blättern sind $G(0,0,0), G(0,0,1), \ldots, G(1,1,1)$, wobei $G(x,y,z) = ((x \wedge \overline{y}) \vee (\overline{x} \wedge z))$.

Beispiel:
Wir wollen Algorithmen finden für die Aufgabe, bei einem 2-Personen-Spiel mit vollständiger Information (z.B. bei Schach, Go, Dame, Mühle, Tic-Tac-Toe) einen optimalen Zug zu finden. Betrachten wir eine gegebene Spielsituation in einem solchen Spiel; der Spieler Weiß sei am Zug; er hat endlich viele Zugmöglichkeiten, welche jeweils eine neue Spielkonfiguration ergeben, dann ist Schwarz am Zug, usw. Nach endlich vielen Spielzügen wird eine Situation erreicht, in der entweder Weiß oder Schwarz gewonnen hat (oder evtl. auch Unentschieden). Diese Betrachtung führt auf sog. Spielbäume, die die möglichen Züge und Folgezüge darstellen:



Hierbei bedeutet 1, dass Weiß gewinnt, und 0, dass Schwarz gewinnt. Man beachte wieder, dass diese Spielbäume nicht wie bei den anderen Graphenalgorithmen explizit gegeben sind, sondern dass sich der Baum implizit aus der (zu analysierenden) Anfangs-Spielstellung für Weiß ergibt.

Die Frage ist nun: kann Weiß, der am Zug ist, so spielen, dass er (auch bei optimalem Spiel von Schwarz) gewinnen kann. Indem man die Nullen und Einsen von den Blättern her zur Wurzel zurückbewertet, kann man diese Frage beantworten: An Knoten, bei denen Weiß am Zug ist, übernimmt man den maximalen Wert der Söhne (also sozusagen eine Oder-Verknüpfung); an Knoten, bei denen Schwarz am Zug ist, übernimmt man den minimalen Wert der Söhne (also sozusagen eine UND-Verknüpfung). Man spricht daher auch von *Min-Max-Bäumen*, die in diesem Kontext wieder nichts anderes als Und-Oder-Bäume sind. Im folgenden Baum sind die sich ergebenden Werte an den inneren Knoten eingetragen.



Das Resultat bei diesem Beispiel ist, dass Weiß gewinnen kann (denn der Wert an der Wurzel ist 1), und zwar, indem er den nach rechts führenden Zug ausführt. (Dieser Und-Oder-Baum entspricht übrigens genau der Schaltung aus Und-Oder-Gattern auf Seite 129).

Der folgende Algorithmus vermag solche Und-Oder-Bäume auszuwerten, wobei wir ausnützen, dass eine Und-Verknüpfung bereits dann den Wert 0 ergibt, wenn einer der Eingänge den Wert 0 hat; analog hat eine Oder-Verknüpfung bereits dann den Wert 1, wenn einer der Eingänge den Wert 1 hat. (Der Einfachheit halber nehmen wir an, dass jeder innere Knoten genau 2 Söhne hat).

Algorithmus 8.4 `oder(`s `:Knoten):boolean`

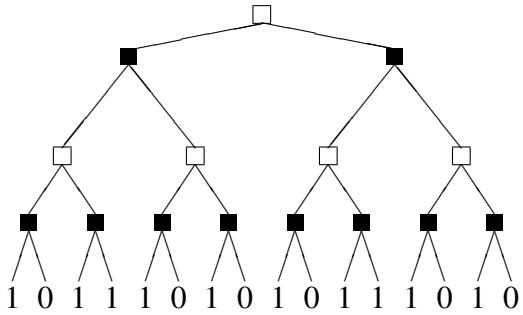
```
1: IF a ist Blatt THEN
2:   return Wert von s
3: ELSE
4:   //seien s_0 und s_1 die beiden Söhne
5:   IF und(s_0) THEN
6:     return TRUE
7:   ELSE
8:     return und(s_1)
```

Algorithmus 8.5 `und(`s `:Knoten):boolean`

```
1: IF s ist Blatt THEN
2:   return Wert von s
3: ELSE
4:   //seien s_0 und s_1 die beiden Söhne
5:   IF ¬oder(s_0) THEN
6:     return FALSE
7:   ELSE
8:     return oder(s_1)
```

Nehmen wir an, der Algorithmus arbeitet auf einem vollständigen Binärbaum der Tiefe t; dieser hat 2^t viele Blätter. Man kann immer eine Beschriftung der Blätter finden, so

dass obiger Und-Oder-Algorithmus *alle* Blätter besuchen muss, so dass die vorgesehene schnelle Auswertung einer Und- bzw. Oder-Verknüpfung nicht eintrifft. Man muss die Knotenbewertung so vorgeben, dass alle linken Äste einer ODER-Verknüpfung den Wert 0 erhalten und alle linken Äste einer UND-Verknüpfung den Wert 1 erhalten (wobei wir annehmen, dass der Algorithmus die Sohn-Knoten von links nach rechts bearbeitet). Die restlichen Wertebelegungen ergeben sich dann zwangsläufig. Das folgende ist ein Beispiel für diesen schlechtesten Fall für $t = 4$.



Das heißt, im schlechtesten Fall muss obiger Und-Oder-Algorithmus alle 2^t Blätter besuchen. Interessanterweise können wir das Verfahren durch eine naheliegende probabilistische Strategie verbessern: wir wählen zufällig aus, welchen Sohn-Wert wir zuerst berechnen. Diese Idee führt auf folgende Prozeduren.

Algorithmus 8.6 `probOder(`s`:Knoten):boolean`

```
1: IF s ist Blatt THEN
2:   return Wert von s
3: ELSE
4:   //seien s_0 und s_1 die beiden Söhne
5:   random k ∈ [0,1]
6:   IF probUnd(s_k) THEN
7:     return TRUE
8:   ELSE
9:     return probUnd(s_{1-k})
```

Algorithmus 8.7 `probUnd(`s`:Knoten):boolean`

```
1: IF s ist Blatt THEN
2:   return Wert von s
3: ELSE
4:   //seien s_0 und s_1 die beiden Söhne
5:   random k ∈ [0,1]
6:   IF ¬probOder(s_k) THEN
7:     return FALSE
8:   ELSE
9:     return probOder(s_{1-k})
```

Jetzt sieht die Sache anders aus. Wenn ein Und auszuwerten ist, welches den Wert 0 hat, so muss mindestens einer der Eingänge 0 sein; und mit Wahrscheinlichkeit 1/2 wird dieser Eingang durch die Zufallsauswahl zuerst ausgewählt. Dann erübrigt sich die Auswertung des zweiten Eingangs. Wenn das Und allerdings den Wert 1 hat, so müssen beide Eingänge ausgewertet werden. Aber in der nächsttieferen Schicht, bei den Oder-Auswertungen, welche in diesem Fall den Wert 1 ergeben, lohnt sich der Probabilismus wieder: mit Wahrscheinlichkeit 1/2 stößt man beim ersten Oder-Eingang bereits auf eine 1, dann braucht der zweite Eingang nicht ausgewertet werden.

Man beachte, dass dieser probabilistische Algorithmus vom „Las Vegas"-Typ ist, denn falsche Ausgaben kann der Algorithmus nicht erzeugen, nur die notwendige Rechenzeit bis zur vollständigen Auswertung des Und-Oder-Baums wird vom Zufall beeinflusst.

Sei $A_0(t)$ der Aufwand (die mittlere Anzahl zu besuchender Blätter), wenn ein Und-Knoten der Stufe t ausgewertet wird, welcher den Wert 0 hat. (Hierbei befinden sich die Blätter auf Stufe 0; für die inneren Knoten bedeutet die „Stufe" t den Abstand von der Blattebene). Analog sei $A_1(t)$ der Aufwand, wenn ein Und-Knoten der Stufe t ausgewertet wird, welcher den Wert 1 hat. Aus Dualitätsgründen ist $A_0(t)$ auch gerade der Aufwand, ein Oder-Gatter auf Stufe t auszuwerten, dessen Wert 1 ist; und $A_1(t)$ ist der Aufwand, ein Oder-Gatter auf Stufe t auszuwerten, dessen Wert 0 ist.

Dann gilt $A_0(0) = A_1(0) = 1$. Für $t > 0$ ergibt sich aufgrund obiger Überlegungen:

$$\begin{aligned} A_0(t) &\leq \tfrac{1}{2} \cdot A_1(t-1) + \tfrac{1}{2} \cdot (A_0(t-1) + A_1(t-1)) \\ &= A_1(t-1) + A_0(t-1)/2 \qquad (1) \\ A_1(t) &= 2 \cdot A_0(t-1) \qquad (2) \end{aligned}$$

Wir setzen (2) in (1) ein und erhalten die folgende Rekursion für A_0:

$$A_0(t) \leq 2 \cdot A_0(t-2) + A_0(t-1)/2$$

Wir versuchen es mit dem Ansatz $A_0(t) \leq \alpha^t$ für eine noch zu bestimmende Konstante α und erhalten hieraus die folgende Gleichung zur Bestimmung von α.

$$\alpha^t = 2 \cdot \alpha^{t-2} + \alpha^{t-1}/2$$

Nach Dividieren mit α^{t-2} ergibt sich die quadratische Gleichung

$$2\alpha^2 - \alpha - 4 = 0$$

welche die (positive) Lösung $\alpha = (1 + \sqrt{33})/4$ hat. Daher ist

$$A_0(t) \leq \left(\frac{1+\sqrt{33}}{4}\right)^t \leq (1.68614)^t$$

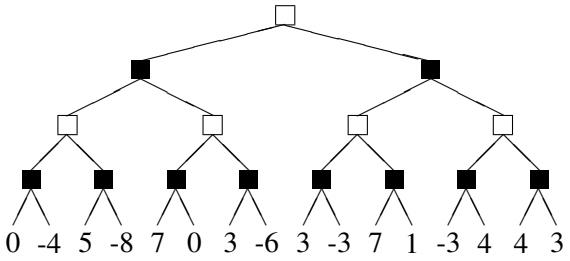
und insgesamt ergibt sich für die Komplexität des Verfahrens

$$\max(A_0(t), A_1(t)) = O\left((1.68614)^t\right) \ll 2^t$$

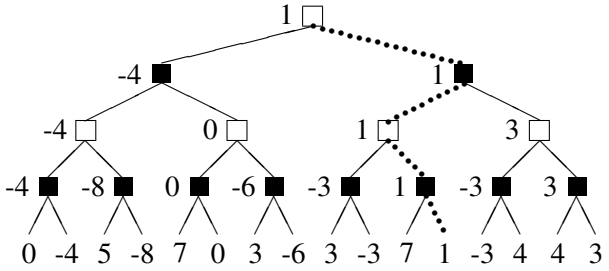
Man kann dieses Ergebnis wieder als Robin-Hood-Effekt bezeichnen, da das beim deterministischen Algorithmus auftretende breite Spektrum vom besten bis zum schlechtesten Fall beim randomisierten Verfahren „eingeebnet" wird, so dass bei allen Eingaben eine mittlere Laufzeit von ca. $(1.7)^t$ garantiert wird. Der mittlere Verzweigungsgrad des zu bearbeitenden Spielbaums wird also (auch bei einer worst-case Eingabe) von 2 auf 1.7 reduziert.

8.4 MiniMax und AlphaBeta

Kehren wir zur Fragestellung zurück, wie man den optimalen Zug bei einem stategischen Spiel wie Schach bestimmt. Es leuchtet ein, dass man in diesem Fall nicht alle Züge und Gegenzüge bis zum Gewinn von Weiß oder Schwarz (oder Remis) durchspielen kann, so dass eine rigorose Durchführung des oben diskutierten Verfahrens illusorisch ist. Was man stattdessen macht, ist den Spielbaum bis zu einer gewissen Tiefe zu entwickeln, um die dann entstandene Spielsituation zu „bewerten“. Als Bewertungskriterium fließt ein: Figurenvorteil, Stellungsvorteil u.a.m. Dies mündet in einer Bewertungszahl, die möglichst realistisch die Gewinnchancen (z.B. aus der Sicht von Weiß) widerspiegeln sollen. Ein realistischer Spielbaum in diesem Sinne (für ein Spiel, bei dem es bei jedem Zug nur 2 Alternativen gibt) könnte also wie folgt aussehen.



Die Rückbewertung von der Blattebene bis hin zur Wurzel erfolgt im Prinzip wie oben beschrieben: Wenn Weiß am Zug ist, so wird der maximale Sohn-Wert ausgewählt, wenn Schwarz am Zug ist, wird der minimale Sohn-Wert ausgewählt. Im folgenden Bild sind alle sich so ergebenden Bewertungen an den inneren Knoten eingetragen. Der optimale Zug für Weiß ist hier derjenige, der die Bewertung 1 ergibt; nämlich der rechte. Gestrichelt eingezeichnet ist die für Schwarz und Weiß optimale Zugfolge.



Der folgende Min-Max-Algorithmus berechnet den Wert an der Wurzel (durch Aufruf von Max(Wurzel)).

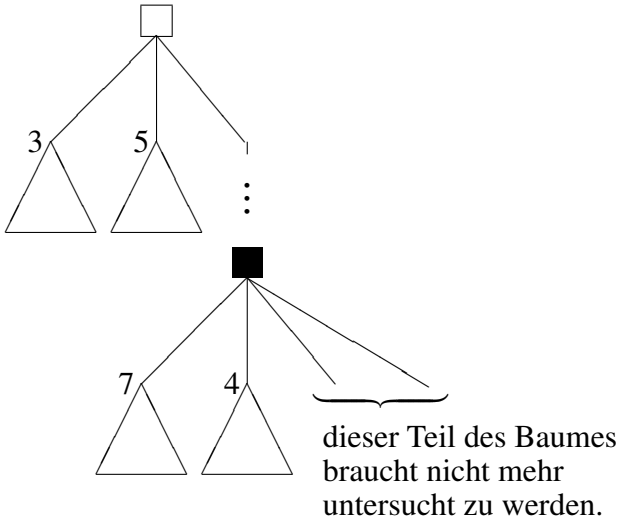
Algorithmus 8.8 `max`$(x$: `Knoten`$)$

```
 1: IF x ist Blatt THEN
 2:    return Bewertung von x
 3: ELSE
 4:    //seien x_1, ..., x_k die Söhne von x
 5:    w = −∞
 6:    FOR i = 1 TO k DO
 7:       v =min(x_i)
 8:       IF v > w THEN
 9:          w = v
10:    return w
```

Algorithmus 8.9 `min`$(x$: `Knoten`$)$

```
 1: IF x ist Blatt THEN
 2:    return Bewertung von x
 3: ELSE
 4:    //seien x_1, ..., x_k die Söhne von x
 5:    w = +∞
 6:    FOR i = 1 TO k DO
 7:       v =max(x_i)
 8:       IF v < w THEN
 9:          w = v
10:    return w
```

Tatsächlich lässt sich dieser Min-Max-Algorithmus in der Effizienz noch verbessern. Betrachten wir folgende Situation während des Ablaufs des Min-Max-Algorithmus:



Die ersten beiden Zugauswertungen ergaben die Werte 3 und 5. Das heißt, der vorläufige Maximumwert an der Wurzel ist 5. (Das tatsächliche Maximum kann noch ≥ 5 sein). Im

weiteren Verlauf wird in einer tieferen Ebene ein Minimum berechnet. Zunächst ergab sich bei den Söhnen der Wert 7, dann der Wert 4; das vorläufige Minimum ist also 4. Das tatsächliche Minimum wird ≤ 4 sein. Egal wie dieses tatsächliche Minimum ausfällt, es wird die Maximumbildung weiter oben nicht mehr beeinflussen, da $5 \geq 4$. Wir können daher auf die Bearbeitung der weiteren Söhne bei der Minimum-Bildung verzichten und können in die nächsthöhere Ebene zurückkehren, indem wir uns mit dem Minimum-Wert 4 zufrieden geben.

Ein analoges Argument zeigt, dass eine Maximumbestimmung in einer tieferen Ebene beendet werden kann, wenn das vorläufige Maximum hierbei $\geq$ einem vorläufigen Minimum aus einer höheren Ebene ist.

Um dieses Verfahren zu implementieren, müssen wir den rekursiven Prozeduren das vorläufig erreichte Maximum und Minimum aus der höheren Ebene als Parameter mitübergeben; diese Werte heißen der α-Wert und der β-Wert und das nachfolgende Verfahren nennt sich *Alpha-Beta-Prozedur*.

Algorithmus 8.10 `alphaBetaMax`$(x$`:Knoten`$, \alpha, \beta$`:Integer`$)$

```
1:  IF x ist Blatt THEN
2:    return Bewertung von x
3:  ELSE
4:    //seien x_1, ..., x_k die Söhne von x
5:    w = α
6:    FOR i = 1 TO k DO
7:      v = alphaBetaMin(x_i, w, β)
8:      IF v > w THEN
9:        w = v
10:     IF w ≥ β THEN
11:       return w //vorzeitiger Rücksprung
12:   return w
```

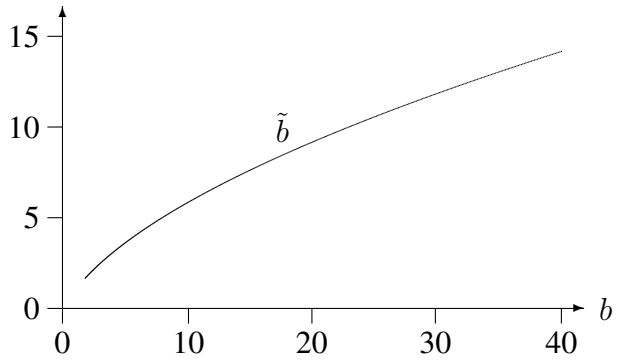
Algorithmus 8.11 `alphaBetaMin`$(x$`:Knoten`$, \alpha, \beta$`:Integer`$)$

```
1:  IF x ist Blatt THEN
2:    return Bewertung von x
3:  ELSE
4:    //seien x_1, ..., x_k die Söhne von x
5:    w = β
6:    FOR i = 1 TO k DO
7:      v = alphaBetaMax(x_i, α, w)
8:      IF v < w THEN
9:        w = v
10:     IF w ≤ α THEN
11:       return w //vorzeitiger Rücksprung
12:   return w
```

Zur Bestimmung des Wertes eines Spielbaumes wird im Hauptprogramm *AlphaBetaMax*(Wurzel,$-\infty$,∞) aufgerufen.

Angenommen, der Verzweigungsgrad (branching factor) des Spielbaums (also die Anzahl der Nachfolger jedes inneren Knotens) sei b und die Zahlenwerte an den Blättern werden zufällig gezogen. Der Spielbaum besitzt bei Tiefe t also b^t Blätter. Wenn man den Spielbaum mittels der Alpha-Beta-Prozedur auswertet, wie groß ist dann der mittlere Verzweigungsgrad des Teilbaums, der die tatsächlich besuchten Knoten darstellt? Aus einem Vergleich dieses gesuchten Wertes mit b ergibt sich, wie effektiv AlphaBeta arbeitet.

Wir zitieren das folgende Resultat: Der mittlere Verzweigungsgrad von AlphaBeta bei einem zufällig bewerteten Spielbaum ist (asymptotisch für große t) $\tilde{b} = \alpha_b/(1-\alpha_b)$. Dabei ist α_b die (positive) Lösung der Gleichung $x^b + x = 1$. Für Werte bis etwa $b = 1000$ wird diese Funktion sehr gut approximiert durch $\tilde{b} \approx 0.925 \cdot b^{0.747}$. Das folgende Diagramm zeigt diese Funktion:



Beispiel: Der typische Verzweigungsgrad (also die Anzahl der zulässigen Züge) bei Schach beträgt typischerweise 35. Der Min-Max-Algorithmus müsste also bei einem zu analysierenden Spielbaum der Tiefe t 35^t Blätter besuchen. Bei Einsatz der Alphabeta-Prozedur besagt das obige Ergebnis, dass der mittlere Verzweigungsgrad nur etwa 13 beträgt. Auflösen der Gleichung $35^t = 13^{t'}$ ergibt $t' = 1.38 \cdot t$. Das heisst, bei gleicher Rechenzeit kann mittels Alphabeta ein ca. 38 Prozent tieferer Spielbaum als bei Minimax bearbeitet werden.

Wir erwähnen noch verschiedene Möglichkeiten der Effizenzverbesserung, die in konkreten Implementierungen von AlphaBeta (zum Beispiel bei Schachprogrammen) eingesetzt werden.

Als Erstes ist es nicht notwendig, so wie wir es hier getan haben, dass die beiden Prozeduren für das Bestimmen des Minimums (Züge von Schwarz) und des Maximums (Züge von Weiß) voneinander getrennt sind und sich gegenseitig aufrufen. Tatsächlich kann man dies mit einer einzigen rekursiven Prozedur realisieren, indem man die Minimumsbildungen auf Maximumsbildungen mittels $\min(x, y) = -\max(-x, -y)$ zurückführt.

Um den Effekt des „tree pruning“, der von der AlphaBeta-Prozedur erreicht wird, möglichst gut auszunützen, empfiehlt es sich, in jeder Prozedur-Inkarnation die Reihenfolge der betrachteten Züge so vorzusortieren, dass diejenigen Züge, die vermutlich die Besten

sind, zuerst betrachtet werden. Auf diese Art wird der mittlere Verzweigungsgrad von AlphaBeta noch weiter reduziert.

Eine weitere Reduktion des mittleren Verzweigungsgrads ist auf folgende Weise möglich. Die Prozedur AlphaBetaMax wird mit den Parametern $\alpha = -\infty$ und $\beta = \infty$ gestartet. Im weiteren Verlauf der rekursiven Prozedur wird dieses „Suchfenster“ $[-\infty, \infty]$ immer enger, bis es auf den Wert, der sich an der Wurzel ergibt, zusammenfällt. Der Effekt des „tree pruning“, der von der AlphaBeta-Prozedur erreicht wird, wird umso besser, je enger das Suchfenster ist, mit dem AlphaBeta aufgerufen wird. Daher lohnt es sich, mit zunächst einer einfachen und effizienten Methode eine Schätzung s für Wert an der Wurzel zu berechnen, und dann erst AlphaBeta für die genauere Berechnung aufzurufen, wobei ein Suchfenster der Form $[s - \varepsilon, s + \varepsilon]$ verwendet wird.

8.5 AVL-Bäume

Ein Suchbaum ist eine Datenstruktur, bei der die Datenobjekte den Knoten eines Binärbaumes entsprechen und anhand von Schlüsseln (Zahlen) aufgefunden werden. Im gesamten Suchbaum muss gelten: die Zahlen im linken Teilbaum sind kleiner als die Wurzel, welche wiederum kleiner ist als die Zahlen im rechten Teilbaum. Soll ein neues Datenobjekt eingefügt werden, so sucht man, bei der Wurzel beginnend, dasjenige Blatt auf, an das das neue Objekt dann als linker bzw. rechter Sohn angefügt wird. Etwas schwieriger ist das Löschen: Ist der zu löschende Knoten ein Blatt, so kann dieses ohne Weiteres entfernt werden. Ist der zu löschende Knoten ein innerer Knoten, dann existiert also ein linker (bzw. rechter) Teilbaum. Anstatt den Knoten selbst zu löschen, überschreibt man diesen mit dem rechtesten Blatt im linken Teilbaum (bzw. dem linkesten Blatt im rechten Teilbaum). Dadurch bleibt die Suchbaum-Eigenschaft erhalten. Ein vollständiger aufgefüllter Binärbaum mit n Knoten hat maximale Pfadlänge $\log n$ (und benötigt damit die Such-, Einfüge- und Löschzeit $\mathcal{O}(\log n)$).

Durch viele Einfüge- bzw. Löschaktionen kann ein Suchbaum „entarten“ - im Extremfall bis zu einer linearen Kette - so dass die Suchzeit bis auf $\mathcal{O}(n)$ anwächst. Daher benötigt man Maßnahmen, die einen Baum, der droht, aus der „Balance“ zu geraten, wieder umzustrukturieren (zu rebalancieren), so dass die Suchzeit $\mathcal{O}(\log n)$ gewährleistet bleibt. Gleichzeitig muss aber auch die Suchbaum-Eigenschaft erhalten bleiben. Mittels *AVL-Bäumen* kann dieses erreicht werden (benannt nach den Anfangsbuchstaben ihrer beiden russischen Erfinder: Adelson-Velskii und Landis).

Es handelt sich um sog. *höhenbalancierte* Bäume. Es wird bei jeder Einfüge bzw. Lösch-Aktion durch evtl. Umstrukturieren des Baumes erreicht, dass für jeden Knoten im Baum gilt:

$$|\text{(Tiefe des linken Teilbaums)} - \text{(Tiefe des rechten Teilbaums)}| \leq 1$$

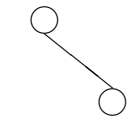
Mit

$$b(k) = \text{(Tiefe des linken Teilbaums von } k) - \text{(Tiefe des rechten Teilbaums von } k)$$

bezeichnen wir die *Balance* des Knotens k. Es gilt aufgrund obiger Forderung immer $b(k) \in \{-1, 0, +1\}$ In jedem Knoten des AVL-Baumes sind somit der Inhalt, die beiden

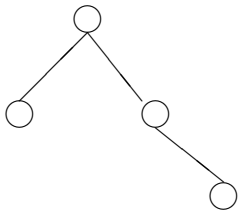
Zeiger auf den linken und rechten Kind-Knoten und die Balance gespeichert. Die folgende Bildfolge zeigt AVL-Bäume der Baumtiefe $n = 0, 1, 2, 3$, die so "unbalanciert" sind, wie es aufgrund der AVL-Eigenschaft noch eben möglich ist:

$B_0 =$

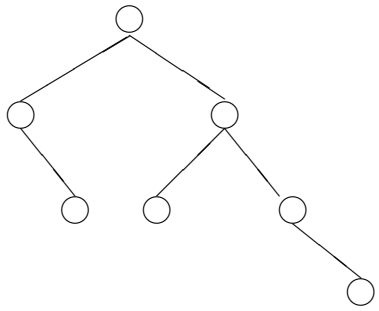


$B_1 =$

$B_2 =$

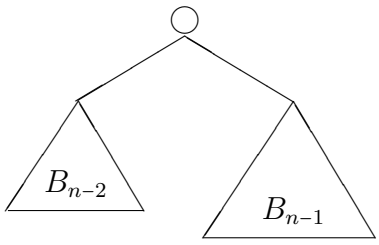


$B_3 =$



Das allgemeine Bildungsgesetz für B_n lautet:

$B_n =$



Wir erhalten somit folgenden Zusammenhang zwischen Tiefe t und Anzahl der Knoten $k = k(t)$ eines AVL-Baumes im schlechtesten Fall:

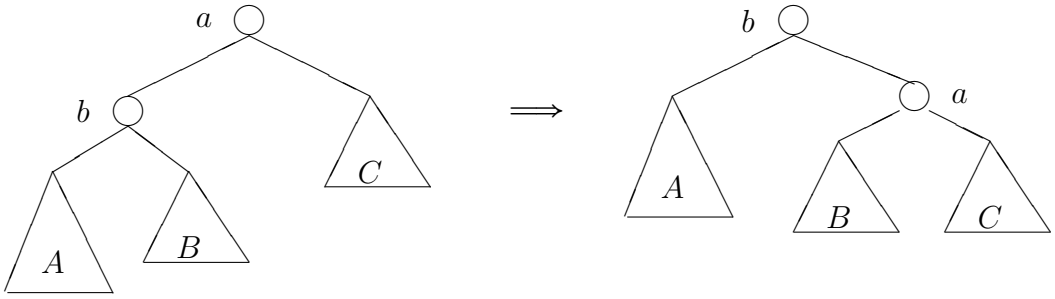
$$k(t) = 1 + k(t-1) + k(t-2)$$

mit $k(0) = 1$ und $k(1) = 2$. Durch Induktion lässt sich leicht zeigen, dass für $t \geq 2$ gilt:

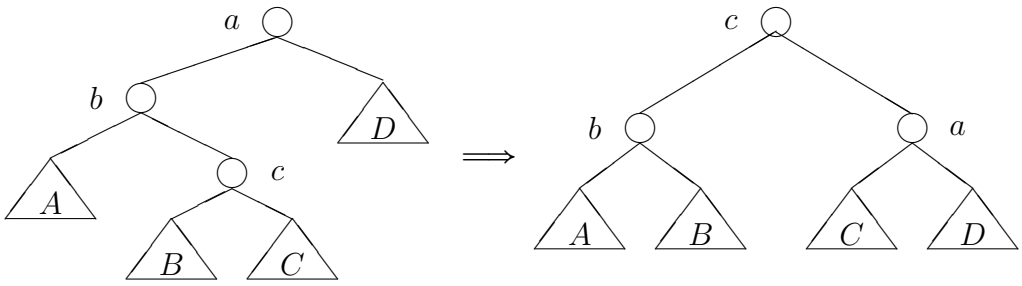
$k(t) \geq 2^{t/2}$. Daraus folgt $t \leq 2 \cdot \log_2 k$. Das heißt, die Baumtiefe ist logarithmisch in der Knotenzahl beschränkt.

Wie wird nun in einen AVL-Baum unter Einhaltung der AVL-Bedingung eingefügt? Zunächst wird der neue Knoten wie in einen gewöhnlichen Suchbaum an der entsprechenden Stelle als neues Blatt eingefügt. Nun können sich entlang des Pfades von diesem Blatt ausgehend bis zur Wurzel die Balance-Werte geändert haben. Diese Werte müssen also geändert werden. Hierbei kann sich jedoch ergeben, dass der neue Balancewert nicht mehr zulässig ist (d.h. +2 oder –2). Dann muss der Baum umstrukturiert werden, um die Balance in allen Knoten wieder herzustellen. Hier treten zwei mögliche Fälle auf (plus ihre spiegelbildlichen Versionen, also eigentlich 4 Fälle).

Fall 1: Der Suchweg verlief über Knoten b nach a. Bei Knoten a ist der linke Teilbaum zu tief (es wurde also in Teilbaum A eingefügt). Hier hilft die sog. *Rotation*:



Fall 2: Der Suchweg verlief über die Knoten c, b nach a. Bei a ist die Balance nicht mehr gegeben, da der linke Teilbaum zu tief ist. (Eingefügt wurde in Teilbaum B oder C). Jetzt hilft die *Doppelrotation*:



Da diese Umstrukturierungsmaßnahmen immer nur entlang des Pfads von dem neu eingefügten Knoten bis zur Wurzel erfolgen müssen, und da die Tiefe eines AVL-Baums höchstens $\mathcal{O}(\log n)$, n = Knotenzahl, ist, ist die Komplexität für eine Einfügeoperation, genau wie die Komplexität der Suchoperation, mit $\mathcal{O}(\log n)$ beschränkt.

Das Löschen in einem AVL-Baum ist etwas komplizierter. Auch hier gilt jedoch, dass nur entlang eines Pfades umstrukturiert werden muss, dass also die Löschoperation gleichfalls die Komplexität $\mathcal{O}(\log n)$ hat.

Kapitel 9

String Matching

Wir betrachten folgende, sehr häufig vorkommende Aufgabe: Gegeben seien ein *Text* $T[1..n]$ und ein *Muster* oder *Pattern* $P[1..m]$ mit $m \leq n$. Die Elemente von T bzw. P entstammen hierbei einem vorgegebenen Alphabet Σ. Wir sagen, dass das Muster P im Text T mit *Shift* s vorkommt, falls gilt:

$$T[s+1, \ldots, s+m] = P[1, \ldots, m]$$

Die Aufgabe besteht darin, alle Shifts s auszugeben, so dass P in T mit Shift s vorkommt. (Einen solchen Shift nennen wir im Folgenden auch einen *zulässigen Shift*).

Eine Variante der Aufgabe bestünde darin, nur den *ersten* zulässigen Shift s zu finden.

Der naive Algorithmus für die obige Aufgabe arbeitet wie folgt:

Algorithmus 9.1 `naiveStringMatch`(s)

```
1: FOR s = 0 TO n − m DO
2:   IF text(s) THEN
3:     output(s)
```

wobei

Algorithmus 9.2 `test`(s)

```
1: FOR j = 1 TO m DO
2:   IF P[j] ≠ T[s + j] THEN
3:     return FALSE
4: return TRUE
```

Die Komplexität von diesem Algorithmus ist $\mathcal{O}((n-m+1) \cdot m) = \mathcal{O}(n \cdot m)$. Die Frage ist, ob es auch Algorithmen gibt, die mit linearem Aufwand bezogen auf die Eingabelänge arbeiten, also mit Komplexität $\mathcal{O}(n+m)$. Wir werden solche Algorithmen in den nächsten Abschnitten vorstellen.

9.1 Rabin-Karp-Algorithmus

Die Idee bei diesem Algorithmus ist, zunächst das Pattern P mittels einer Hashfunktion auf ein wesentlich kleineres Wort (oder Zahl) $h(P)$ abzubilden, so dass dieses in eine Speicherzelle passt (also mit Komplexität $\mathcal{O}(1)$ verarbeitet werden kann). Im weiteren Verlauf des Matching-Vorgangs werden anstelle der Originaltextabschnitte der Länge m deren Hashwerte mit $h(P)$ verglichen. Sollten die Hashwerte nicht übereinstimmen, so kann unmittelbar das Vergleichsfenster weitergeschoben werden. (Man spricht anschaulich auch davon, dass von dem Pattern ein „Fingerabdruck“ genommen wird; also eine vergleichsweise kurze Information, die jedoch das Pattern (nahezu) eindeutig identifiziert.

Nur wenn die Hashwerte übereinstimmen, so müssen der entsprechende Textabschnitt doch Zeichen für Zeichen mit P verglichen werden, da eine Kollision mit einem Text ungleich P stattgefunden haben kann. Wir können also nicht unbedingt sicher sein, dass aus $h(P) = h(T[s+1..s+m])$ folgt $P = T[s+1..s+m]$.

Ein weiterer Trick besteht darin, dass das Neu-Berechnen der Hashfunktion nach Verschieben des zu betrachtenden Textfensters mit Aufwand $\mathcal{O}(1)$ geschehen kann, wenn man eine Hashfunktion nach der Kongruenzmethode verwendet.

Beispiel:
Sei $\Sigma = \{0,\ldots,9\}$ das zugrundeliegende Alphabet. Soll beim laufenden Text T = ...12345678... das Fenster der Größe $m = 6$ von 123456 nach 234567 weitergeschoben werden, so können wir annehmen, dass $t = h(123456) = 123456 \bmod q$ bereits berechnet wurde. Um $t' = h(234567) = 234567 \bmod q$ zu bestimmen, rechne man nur $t' = ((t - 1 \cdot u) * 10 + 7) \bmod q$, wobei $u = 10^{m-1} \bmod q$. Das heißt, die Ziffer 1 vorne wird entfernt; der Rest mit der Basiszahl 10 multipliziert, und dann hinten die neue Ziffer 7 angefügt. Alle Zwischenrechnungen können modulo q gerechnet werden.

Der folgende Algorithmus führt dies aus für ein allgemeines Alphabet Σ, so dass $|\Sigma|$ nun die Rolle der Zahl 10 übernimmt. Sei q eine von vorneherein festgelegte (oder zufällig gewählte) Zahl, die die weiter unten im Text diskutierten Bedingungen erfüllt.

Algorithmus 9.3 `rabin-Karp`(P,T,n,m)

```
 1: p=0
 2: t=0
 3: u=|Σ|^{m-1} mod q
 4: FOR i = 1 TO m DO
 5:    p := (|Σ| · p + P[i]) mod q
 6:    t := (|Σ| · t + T[i]) mod q
 7:    FOR s = 0 TO n − m DO
 8:       IF p = t THEN
 9:          IF test(s) THEN
10:             output s
11:       IF s < n − m THEN
12:          t := ((t − T[s + 1] · u) · |Σ| + T[s + m + 1]) mod q
```

Die Anforderungen an die Zahl q sind die folgenden: Die Zahl q sollte in eine Speicherzelle passen, so dass alle Rechnungen modulo q mit Aufwand $\mathcal{O}(1)$ ausgeführt werden können. (Besser noch: $|\Sigma| \cdot q$ sollte in eine Speicherzelle passen, da solche Zahlen als Zwischenergnisse – vor der Reduktion modulo q – auftreten können). Ferner empfiehlt es sich – genauso wie bei der Kongruenzmethode (Seite 44) – q als Primzahl zu wählen, um eine gute Streuung der Hashwerte zu erreichen. Andererseits sollte q nicht zu klein gewählt werden, da sonst zu oft Kollisionen zu erwarten sind. Die Zahl q sollte größer als die Patternlänge m sein.

Wenn man die Primzahl q zufällig aus einem festgelegten Zahlenintervall zieht, so befreien wir uns von dem jeweils möglichen schlechtesten Fall.

Im schlechtesten Fall hat der Rabin-Karp-Algorithmus dieselbe Komplexität wie der naive Algorithmus, nämlich $\mathcal{O}(m \cdot n)$. Wenn zum Beispiel $T = a^n$ und $P = a^m$, so ist jeder Shift ein zulässiger Shift, so dass derselbe Aufwand wie beim naiven Algorithmus entsteht.

Es ist aber realistischer anzunehmen, dass das Pattern im Text nicht sehr häufig vorkommt, etwa nur $\mathcal{O}(1)$-mal. In diesem Fall hat der Rabin-Karp-Algorithmus die mittlere Komplexität

$$\mathcal{O}(m) + \mathcal{O}(n) + (\mathcal{O}(1) + (\text{mittlere Anzahl der Kollisionen})) \cdot \mathcal{O}(m)$$

Für die meisten Texte und Pattern wird die Anzahl der Kollisionen in etwa der statistischen Trefferhäufigkeit entsprechen. Das heißt, es sind $n/q \leq n/m$ Kollisionen zu erwarten. Dies eingesetzt erhalten wir damit die mittlere Komplexität $\mathcal{O}(n+m)$.

9.2 String Matching mit endlichen Automaten

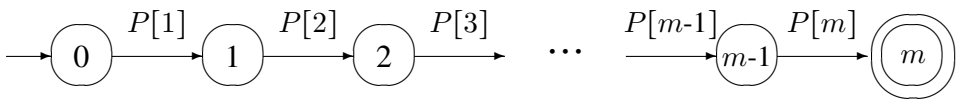
Was den naiven Algorithmus ineffizient macht, ist dass bei einer aufgefundenen Nicht-Übereinstimmung (einem „mismatch“) zwischen dem Pattern P und dem Textfenster $T[s+1..s+m]$ dieses Textfenster nur um eine Position weitergeschoben wird. Evtl. lässt sich aus dem bereits gelesenen Teil von $T[s+1..s+m]$ schließen, dass ein größerer Shift möglich ist als 1.

Diese Idee kann mittels endlicher Automaten realisiert werden. Anhand des Patterns $P = P[1..m]$ wird ein endlicher Automat konstruiert. Dieser hat $m+1$ Zustände $0, 1, \ldots, m$, wobei 0 der Startzustand ist und m der Endzustand, welcher signalisiert, dass das Pattern erkannt wurde, also dass ein zulässiger Shift aufgefunden wurde.

Der Automat enthält die folgenden Übergänge, die dafür sorgen, dass bei Lesen des Patterns Schritt für Schritt in den Endzustand übergegangen wird:

$$\delta(i-1, P[i]) = i \quad (i = 1, 2, \ldots, m)$$

Skizze:

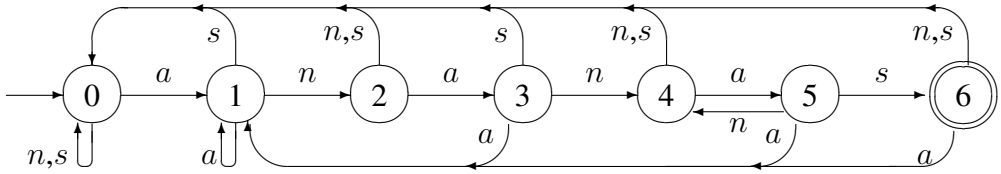


Dies nennen wir den *Skelettautomaten*. Dieser ist natürlich noch kein vollständig beschriebener endlicher Automat, denn es fehlen noch die Übergänge $\delta(i-1,a)$ für $a \neq P[i]$, $i = 1,\ldots,m$ und die Übergänge vom Zustand m aus. Diese Übergänge entsprechen gerade der Situation bei einem mismatch. Und zwar muss dabei berücksichtigt werden, dass ein Suffix von $P[1]\ldots P[i-1]a$ wieder ein Präfix des Patterns sein kann, und daher muss zu einem entsprechenden Zustand, der diesen partiellen match repräsentiert, zurückgesprungen werden.

Die entsprechende Definition ist die folgende: Es gilt

$$\delta(i,a) = \begin{cases} \max\{k \leq i \mid P[1..k] \text{ ist Endstück von } P[1..i]a\}, \\ \quad \text{falls das Maximum existiert} \\ 0, \quad \text{sonst} \end{cases}$$

Beispiel:
Das Pattern sei *ananas* und $\Sigma = \{a,n,s\}$. Dann ergibt sich der folgende Automat.



Beispielsweise gilt der Übergang $\delta(5,n) = 4$, weil das Wort $P[1..4] = anan$ ein Endstück von $P[1..5]n = ananan$ ist. (Das Wort $P[1..2] = an$ wäre zwar auch ein Endstück; jedoch ist das *maximale* k auszuwählen).

Der Algorithmus läuft nun so ab, dass zunächst anhand des Patterns der entsprechende endliche Automat aufgebaut wird, und dann der Text mit Hilfe des endlichen Automaten Symbol für Symbol verarbeitet wird. Jedesmal, wenn der Automat in den Endzustand gerät, wird ein zulässiger Shift gemeldet.

Algorithmus 9.4 `stringAutomat`(T,n,m)

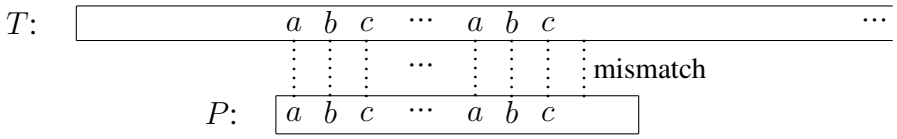
```
1: //Konstruiere den endlichen Automaten
2: z = 0
3: FOR i = 1 TO n DO
4:     z = δ(z, T[i])
5:     IF z = m THEN
6:         output i − m
```

Dieser Algorithmus arbeitet in der Zeit $\mathcal{O}(|\Sigma|m + n)$. Hierbei wird zur Konstruktion der Übergangstabelle des endlichen Automaten schon ein recht kleverer Algorithmus vorausgesetzt, wenn dies in der Zeit $\mathcal{O}(|\Sigma|m)$ (also linear in der Größe der Automaten-Übergangstabelle) gehen soll. Wir verzichten auf eine explizite Angabe eines solchen Algorithmus, da die in den nächsten Abschnitten besprochenen Algorithmen noch effizienter sind.

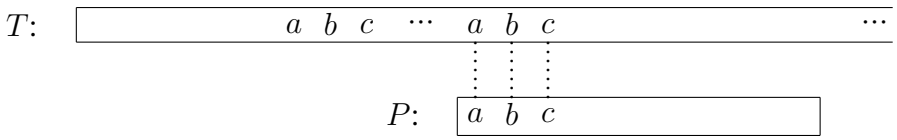
9.3 Knuth-Morris-Pratt Algorithmus

Beim String-Matching mit endlichen Automaten ist das Erstellen der Automaten-Übergangstabelle recht aufwändig. Es ist eine zweidimensionale Tabelle mit $|\Sigma| \cdot (m+1)$ Einträgen erforderlich. Der folgende Algorithmus von Knuth-Morris-Pratt verwendet konzeptionell dieselbe Idee, nur wird stattdessen eine weniger aufwändige, eindimensionale Tabelle mit m Einträgen verwendet. Diese „Verschiebetabelle“ beruht allein auf der Patterninformation und kann in linearer Zeit in der Patternlänge konstruiert werden.

Nehmen wir an, ein Teil des Patterns wurde bereits mit dem Text verglichen, bis ein mismatch auftritt:



Bei diesem Beispiel kann das Pattern im nächsten Schritt soweit verschoben werden, bis der Teil „abc“ wieder zur Deckung gebracht wird. Dazwischenliegende Verschiebepositionen können ausgeschlossen werden. Ferner brauchen in diesem Fall die ersten 3 Zeichen des Patterns nicht noch einmal mit dem Text verglichen zu werden.



Wesentlich ist hier, dass der Teil des Patterns vor der mismatch-Stelle identisch ist mit einem Anfangsstück des Patterns. Was wir beim Knuth-Morris-Pratt Algorithmus zunächst berechnen, ist die folgende Verschiebetabelle $\Pi[1..m]$ mit

$$\Pi[q] \;=\; \max\{k < q \mid P[1..k] = P[q-k+1..q]\} \quad (q = 1, \ldots, m)$$

Beispiel:
Für das Pattern $ananas$ ergibt sich folgende Tabelle:

i	1	2	3	4	5	6
P	a	n	a	n	a	s
Π	0	0	1	2	3	0

Wir verschieben zunächst das Problem, die Verschiebetabelle zu berechnen. Der eigentliche Algorithmus arbeitet wie folgt:

Algorithmus 9.5 KMP(P, T, n, m)

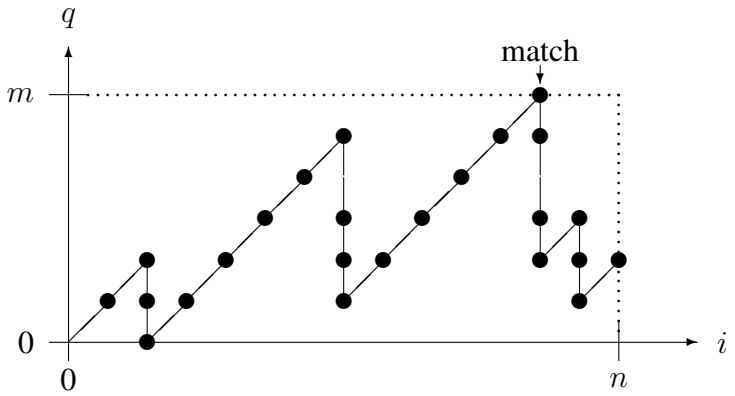
```
1:  //Berechne die Verschiebetabelle Π[1..m]
2:  q = 0
3:  FOR i = 1 TO n DO
4:    WHILE (q > 0) ∧ (P[q + 1] ≠ T[i]) DO
5:      q = Π[q]
6:    IF P[q + 1] = T[i] THEN
7:      q = q + 1
8:    IF q = m THEN
9:      output i − m
10:     q = Π[q]
```

Hierbei ist i die aktuelle, zu vergleichende Textposition; q ist die Zahl der Positionen, in denen momentan das Pattern und der Text übereinstimmen; m ist die Patternlänge; n ist die Textlänge.

In der WHILE-Schleife wird im Falle eines mismatches (nämlich $(P[q+1] \neq T[i])$) der Wert von q anhand der Verschiebetabelle reduziert ($q := \Pi[q]$), danach wird die fragliche Pattern-Position $q+1$ wieder mit dem Text verglichen, usw. Wenn schließlich Übereinstimmung zwischen $P[q+1]$ und $T[i]$ festgestellt wird, kann das „Fenster“ um eine Position erweitert werden (also $q := q+1$ und $i := i+1$). Ein match tritt genau dann auf (und wird entsprechend ausgegeben), wenn q den Wert m erreicht.

Zur Komplexitätsanalyse: Programmtechnisch wurden oben zwei ineinander verschachtelte Schleifen verwendet, wobei die äußere von 1 bis n läuft, und die innere, die WHILE-Schleife, schlimmstenfalls den Wert von q von m in Einerschritten bis auf auf 0 reduziert. Dass die Gesamtkomplexität beider Schleifen trotzdem nicht $\mathcal{O}(nm)$ ist, sondern nur $\mathcal{O}(n)$, zeigt eine Amortisationsanalyse. Der wesentliche Punkt hierbei ist, wenn die innere WHILE-Schleife den q-Wert mal stark reduziert hat, so muss dieser erst wieder in Einerschritten aufgebaut werden. Das hat den Effekt, dass ein solcher schlechtester Fall der inneren WHILE-Schleife nur sehr selten auftreten kann.

Das folgende Diagramm zeigt, wie sich der q-Wert typischerweise im Verlauf der beiden Schleifen verändert.



Jeder Punkt auf der Kurve entspricht einer Veränderung des q-Wertes und macht einen

$\mathcal{O}(1)$ Anteil an der Gesamtkomplexität des Knuth-Morris-Pratt Algorithmus aus (abgesehen von dem Aufwand zur Berechnung der Verschiebetabelle $\Pi[1..m]$). Es genügt also, die Anzahl der q-Wertveränderungen im Verlauf der beiden Schleifen abzuschätzen. Jeder Punkt, der auf einer aufwärts gerichteten Linie sitzt, entspricht einer Ausführung von $q := q + 1$ (und $i := i + 1$). Solche Punkte gibt es insgesamt n Stück, da i die Werte von 1 bis n durchläuft. Die abwärts laufenden Linien entsprechen den WHILE-Schleifendurchläufen. Jede abwärts laufende Linie kann höchstens so viele Punkte enthalten, wie auf der davorliegenden aufwärts laufenden Linie sitzen. Also gilt, dass die Anzahl der „Abwärts-Punkte“ höchstens so groß ist wie die Anzahl der „Aufwärts-Punkte“. Somit ist die Anzahl der Punkte insgesamt durch $2n$ beschränkt. Somit haben die beiden ineinandergeschachtelten Schleifen eine Komplexität von $\mathcal{O}(n)$.

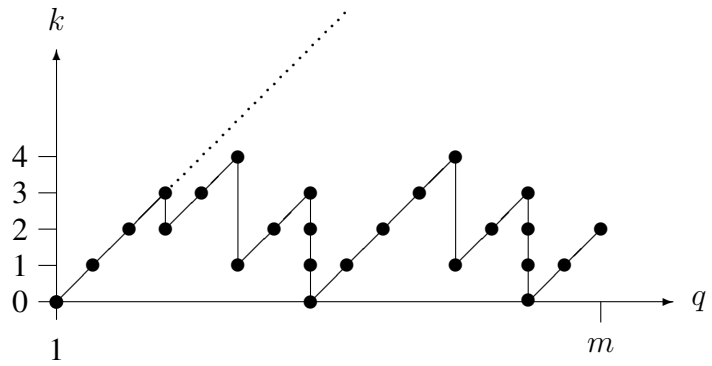
Als Nächstes betrachten wir den Algorithmus zum Aufbau der Verschiebetabelle. Dieser ist konzeptionell genauso aufgebaut wie der vorige Algorithmus, nur wird hier nicht das Pattern mit dem Text verglichen, sondern das Pattern mit sich selbst.

Algorithmus 9.6 `buildΠ`(P, m)

```
1: Π[1] = 0
2: k = 0
3: FOR q = 2 TO m DO
4:    WHILE (k > 0) ∧ (P[k + 1] ≠ P[q]) DO
5:       k = Π[k]
6:    IF P[k + 1] = P[q] THEN
7:       k = k + 1
8:    Π[q] = k
```

Die Begründung zur Korrektheit dieses Programms ist analog zu der zuvor gegebenen. Hier ist q die aktuelle Patternposition, während k die Position des Anfangsabschnitts des Patterns ist, welche potenziellerweise mit dem Endstück von $P[1..q]$ übereinstimmt. Ausgehend von $\Pi[1] = 0$ lassen sich die weiteren Π-Werte mit Hilfe der bereits berechneten Π-Werte bestimmen: Sei $q > 1$ und sei k solcherart maximal gewählt, dass $P[1 .. + 1] = P[q - k .. q]$. (Der Wert von k wird durch die WHILE-Schleife, basierend auf den bereits berechneten Π-Werten, bestimmt). Dann ergibt sich $\Pi[q] = k + 1$.

Wir analysieren die Komplexität dieses Verfahrens. Die k-Werte verändern sich – in Abhängigkeit von der q-Schleife – typischerweise so wie im folgenden Diagramm dargestellt. Man beachte, dass der k-Wert höchstens den Wert $q-1$ annehmen kann (entspricht der gestrichelten Linie).



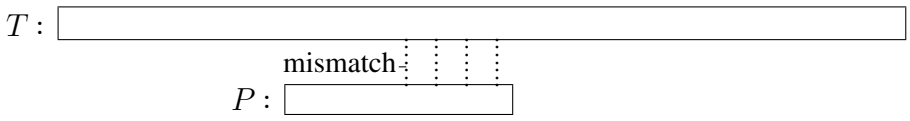
Jeder Punkt in diesem Diagramm entspricht wieder einem $\mathcal{O}(1)$ Anteil an Komplexität. Es gilt also, die maximal mögliche Anzahl der Punkte abzuschätzen. Es gibt höchstens m Punkte, die auf einer aufwärts führenden Linie sitzen, da diese mit einer Erhöhung des q-Wertes einhergehen und q nur die Werte von 1 bis m durchläuft. Für jeden Punkt auf einer abwärts führenden Linie gibt es zuvor einen Punkt auf einer aufwärts gerichteten Linie, der auf derselben Höhe sitzt. Daher kann die Anzahl der Punkte nach oben mit $2m = \mathcal{O}(m)$ abgeschätzt werden.

Insgesamt ergibt sich daher für die Komplexität des Knuth-Morris-Pratt Algorithmus die worst-case Komplexität $\mathcal{O}(m + n)$.

9.4 Boyer-Moore Algorithmus

Wir wollen noch einen weiteren schnellen String-Matching-Algorithmus, allerdings nur skizzenhaft, vorstellen. Grundsätzlich ist hier auch die Idee vorhanden, das Pattern am Text entlangzuschieben, und beim Feststellen eines mismatches das Pattern um einen möglichst großen Abschnitt weiterzuschieben.

Die neue Idee hierbei ist, die Vergleiche zwischen dem Textabschnitt (der Länge m) und dem Pattern nicht wie bisher von links nach rechts, sondern *von rechts nach links* durchzuführen; vgl. folgende Skizze.



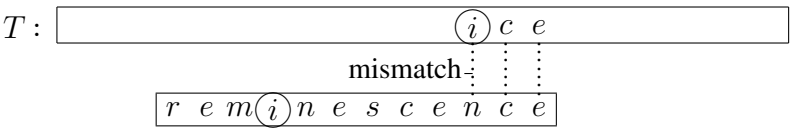
Es ist möglich, dass nach dem Feststellen eines mismatches sehr große Verschiebungen, evtl. bis zur Größenordnung der Patternlänge m, durchgeführt werden können. Im Idealfall (der wohl auch nicht weit vom Durchschnittsfall entfernt ist; dieser ist aber schwer zu analysieren) ergibt sich die folgende Situation: falls Pattern $P[1..m]$ und Textabschnitt $T[s+1..s+m]$ verschieden sind, so kann dies bereits nach $\mathcal{O}(1)$ Vergleichen durch einen mismatch festgestellt werden. Danach ist womöglich eine Verschiebung um $m - o(m)$ Positionen möglich, so dass der Algorithmus (abgesehen von einer Initialisierungsphase, in der gewisse „Verschiebetabellen“ aufgebaut werden müssen) insgesamt nur die

Komplexität $\mathcal{O}(1) \cdot \frac{n}{m-o(m)} = \mathcal{O}(n/m)$ hat. Dies wäre noch schneller als der Knuth-Morris-Pratt Algorithmus und tatsächlich ist der Boyer-Moore Algorithmus in der Praxis der schnellste String-Matching Algorithmus.

Wir müssen noch diskutieren, wie die Verschiebung des Patterns vonstatten geht. Der Boyer-Moore Algorithmus arbeitet mit zwei verschiedenen Strategien; er wählt bei jedem mismatch schließlich diejenige Strategie, die die größte Verschiebung ergibt.

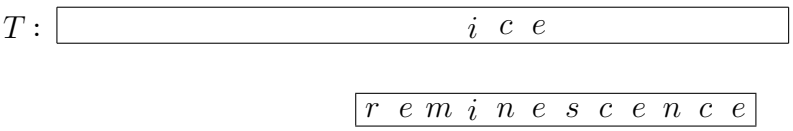
1.) *Die „bad character"-Strategie:*

Nehmen wir an, es gibt beim dritten Vergleich mit dem Pattern $reminescence$ einen mismatch.

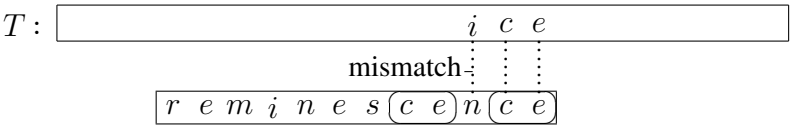


Hierbei ist das „bad character" das Zeichen i im Text. Für den nächsten potenziell möglichen match schauen wir vor der mismatch-Position nach dem am weitesten rechts vorkommenden i im Pattern nach (dieses ist eingekreist). Wir können das Pattern nun so weit verschieben, bis die beiden i's zur Deckung kommen; dazwischen kann definitiv kein match auftreten.

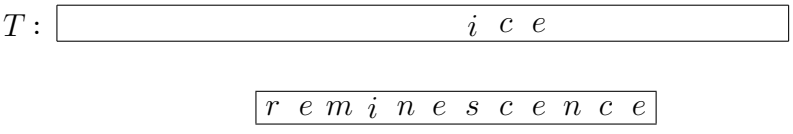
Nach der Verschiebung erhalten wir:



2.) *„good suffix"-Strategie:*



Wir suchen dieses Mal nach Feststellen eines mismatches nach dem am weitesten rechts (vor der mismatch-Position) vorkommenen „good suffix", also derjenigen Zeichenfolge (hier: ce), in der Text und Pattern übereingestimmt haben. Wir können nun das Pattern so weit verschieben, bis diese Textabschnitte zur Deckung gebracht werden:



Für jede der Strategien muss zuvor eine geignete Verschiebetabelle anhand des Patterns in der Zeit $\mathcal{O}(m)$ berechnet werden. Es wird natürlich jeweils gemäß derjenigen Strategie verfahren, die die größere Verschiebung mit sich bringt (im obigen Beispiel die bad character-Strategie). Die Berechnung der Verschiebetabelle für die good suffix-Strategie kann analog dem Knuth-Morris-Pratt Algorithmus erfolgen, allerdings nicht von links nach rechts durch P laufend sondern von rechts nach links.

Obwohl der Boyer-Moore Algorithmus im Mittel sehr schnell ist, ist die worst-case Komplexität allerdings miserabel; sie ist $\mathcal{O}(mn)$, wie beim naiven Algorithmus. Man überprüft leicht, dass der schlechteste Fall zum Beispiel eintritt, wenn Text und Pattern die Form $T = a^n$ und $P = a^m$ haben.

9.5 Suffix-Bäume

Bei den bisherigen Pattern Matching-Algorithmen fand immer, nachdem P und T gegeben waren, eine gewisse Vorverarbeitungsphase statt, die darin bestand, eine geeignete Tabelle oder eine andere Datenstruktur bereitzustellen, wobei diese Datenstruktur aber immer nur vom Pattern P (und dem zugrunde liegenden Alphabet) abhing. Mit Hilfe dieser Datenstruktur ließ sich dann der eigentliche Suchprozess in der Zeit $\mathcal{O}(n)$ erledigen.

In diesem Abschnitt wollen wir den Fall betrachten, dass der zu durchsuchende Text T ein für alle Mal feststeht (und nach verschiedenen Pattern durchsucht werden soll), so dass wir uns daher eine Vorverarbeitungsphase leisten können und wollen, die eine Datenstruktur aufbaut, die nun vom Text T und nicht vom Pattern P abhängt. Der Vorteil wird sein, dass verschiedene Suchanfragen, wie „wo (und wieoft) kommt das Pattern P im Text T vor?“ sehr effizient verarbeitet und beantwortet werden können. Der Aufwand für die Vorverarbeitung, also den Aufbau einer geeigneten Datenstruktur, die T repräsentiert, ist allerdings größer als bei den Algorithmen in den vorigen Abschnitten. Diese Situation des feststehenden Textes kommt typischerweise bei Anwendungen in der Bioinformatik vor; der Text könnte z.B. eine DNA-Sequenz (über dem Alphabet $\{A, T, G, C\}$) sein.

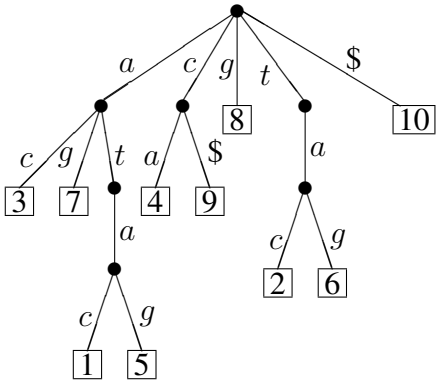
Wir wollen hier die Datenstruktur des *Suffix-Baumes* betrachten (im Englischen *suffix-tree* oder *suffix-trie*, als Wortverquickung von *tree* und re*trie*val). Wir betrachten ein Beispiel. Sei der Text $T = atacatagc\$$ gegeben. Hierbei ist das Alphabet um das Extra-Zeichen \$ angereichert, das das Textende signalisiert (und nur dort vorkommt). Für jede Textposition $i = 1, 2, \ldots, n$ (hier: $n = 10$) notieren wir den kürzesten Teilstring von T, der bei Position i beginnt und nur ein Mal im Text vorkommt. Daher *identifiziert* dieser

Textstring die Position i.

Position	Teilstring
1	$atac$
2	tac
3	ac
4	ca
5	$atag$
6	tag
7	ag
8	g
9	c\$
10	\$

Text T = $atacatagc$\$

Der Suffix-Baum besitzt die Blätter $1, 2, \ldots, n$, wobei der Pfad von der Wurzel bis zum betreffenden Blatt i gerade mit den Buchstaben des identifizierenden Teilstrings markiert ist.



Um zum Beispiel festzustellen, ob der Teilstring at im Text vorkommt, folgt man von der Wurzel her der mit a und dann t beschrifteten Kante und stellt dann fest, dass man sich in einem Teilbaum befindet, der die Blätter 1 und 5 hat. Daher kommt dieser Teilstring an den Positionen 1 und 5 im Text T vor.

Wenn man nach Vorkommen des Pattern-Strings aca sucht, so findet man nach Lesen von ac das Blatt 3 vor; das Teilwort ac des Patterns kommt also lediglich an der Position 3 vor. Jetzt kommt es nur noch darauf an, ob die Fortsetzung (ab Position 5 im Text) auch übereinstimmt (was hier der Fall ist). Dies testet man am Besten ab der betreffenden Text- bzw. Patternstelle durch direkten Vergleich.

Wenn man dagegen nach dem Pattern tg sucht, so stellt man fest, dass kein entsprechender Weg im Baum vorgesehen ist; also ist dieses Pattern nicht im Text vorhanden.

Diese Beispiele zeigen, dass mit Hilfe eines Suffix-Baumes für den Text T die Suche nach Vorkommen eines Pattern P der Länge m in der Zeit $\mathcal{O}(m)$ möglich ist. Darüber hinaus sind mit Hilfe des Suffix-Baumes auch noch komplexere Suchanfragen effizient beantwortbar.

Man beachte, dass in der Literatur zahlreiche Varianten von Suffix-Bäumen existieren. Zum einen kann man Kanten, die nur eine Nachfolgerkante haben, zusammenfassen und

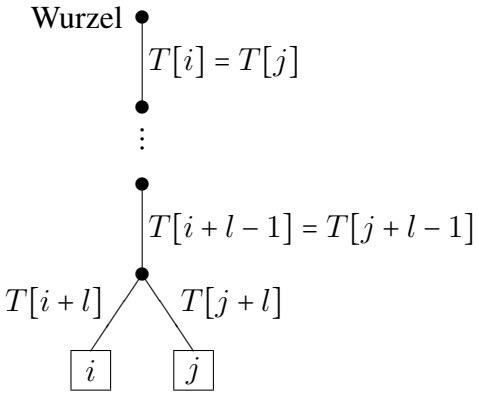
gleichzeitig kann man dann die entsprechenden Symbol-Beschriftungen zu Strings zusammenfassen. Manchmal werden nicht wie hier die *kürzesten* Teilstrings angegeben, die eine Position i eindeutig charakterisieren, sondern stattdessen wird der gesamte Suffix von T, beginnend bei Position i, im Suffix-Baum notiert.

Die Frage ist nun, wie man einen Suffix-Baum für einen gegebenen Text T (effizient) konstruiert. Eine unmittelbar einleuchtende Konstruktion ist die Folgende. Wir konstruieren den Suffix-Baum, indem wir den Text T *von rechts nach links* abarbeiten. Zunächst starten wir mit einem Baum, der nur aus einer Wurzel besteht. Diesen bezeichnen wir mit B_n. In jedem weiteren Schritt wird ein weiterer Textbuchstabe $T[i]$, $i = n, n-1, \ldots, 1$, mit einbezogen und der bisher konstruierte Baum B_i dadurch zu einem neuen Baum B_{i-1} erweitert. Der endgültige Suffix-Baum ist dann B_0.

Die Konstruktion von B_{i-1}, basierend auf B_i, unter Berücksichtigung des Text-Buchstabens $T[i]$, geschieht wie folgt. Startend von der Wurzel folgenden wir den Kanten von B_i mit den Beschriftungen $T[i], T[i+1], \ldots$, bis zum ersten Mal der Fall eintritt, dass keine mit $T[i+k]$ beschriftete Kante vorhanden ist ($k \geq 0$). Nun wird eine mit $T[i+k]$ beschriftete Kante hinzugefügt. Nun sind zwei Fälle zu unterscheiden.

Fall 1: Die neue Kante wurde an einen inneren Knoten von B_i angefügt. In diesem Fall erhält diese Kante ein mit i beschriftetes Blatt und die Konstruktion von B_{i-1} ist fertig.

Fall 2: Die neue Kante wird an ein bisheriges Blatt mit der Nummer $j > i$ angefügt. Dann wird dieses Blatt zu einem inneren Knoten und es müssen ggfs. weitere Kanten angefügt werden, bis zu einer Position, bei der sich die letzten Buchstaben von $T[i..i+l]$ und $T[j..j+l]$ zum ersten Mal unterscheiden ($l \geq k$). Dies führt im Baum B_{i-1} dann zu einem Teilbaum der folgenden Form:



Da bei dieser Konstruktion jeder Textbuchstabe $T[i]$ betrachtet wird, und jeder von diesen evtl. bis zum Textende verfolgt werden muss, ist die worst-case Komplexität dieses Verfahrens $\mathcal{O}(n^2)$. Tatsächlich wird die mittlere Tiefe des Suffix-Baumes aber $\mathcal{O}(\log n)$ sein, so dass die average-case Komplexität des Verfahrens von der Ordnung $\mathcal{O}(n \log n)$ ist. Es wurden in der Literatur verschiedene Verbesserungen vorgeschlagen, die im worst-case sogar den Aufwand $\mathcal{O}(n)$ erreichen.

Kapitel 10

Heuristische Algorithmen

Unter heuristischen Algorithmen verstehen wir eine Sammelbezeichnung für Algorithmen, die versuchen, eine (möglichst optimale) Lösung in einem (exponentiell) großen Lösungsraum durch eine durch „Heuristiken“ gesteuerte Suche zu finden. Heuristiken sind hierbei problem-spezifische Informationen, die es ermöglichen, evtl. schneller zu einer (optimalen, oder wenigstens passablen) Lösung zu gelangen, als durch „blindes“ Suchen, also vollständiges Aufzählen aller potenziellen Lösungen. Im Allgemeinen ist es nicht möglich, eine Garantie abzugeben in Bezug auf die erreichbare Lösungsgüte (Performanz) und/oder in Bezug auf die Laufzeit. An Stelle einer rigorosen Komplexitätsanalyse tritt oft eine systematische *experimentelle Algorithmenanalyse*, welche oftmals aufzeigt, dass diese Algorithmen ein erstaunlich gutes Verhalten haben (obwohl dies von theoretischer Seite nicht vollständig begründet, also bewiesen, werden kann).

Wir wollen im Folgenden verschiedene Ansätze, die in diese Kategorie der heuristischen Algorithmen fallen – aber untereinander z.T. sehr unterschiedlich sind – diskutieren. Als Anwendungsbeispiel, an dem wir die verschiedenen Konzepte demonstrieren, haben wir das (NP-vollständige) Traveling Salesman Problem (TSP) gewählt, das in Abschnitt reftsp und 8.2 eingeführt wurde

Die heuristischen Verfahren lassen sich grob in zwei Klassen einteilen: Erstens, solche Verfahren, die eine Lösung sukzessive, ausgehend von der leeren Lösung, konstruieren. Zweitens, solche Verfahren, die bereits gegebene (evtl. zufällig hergestellte) Lösungen versuchen zu verbessern. Im Abschnitt 10.2 wird ein Verfahren der ersten Kategorie behandelt und die Abschnitte 10.3, 10.4, 10.5 enthalten Verfahren, die in die zweite Kategorie fallen. Die Algorithmen der zweiten Kategorie sind meist probabilistisch. In Abschnitt 10.1 wird ein weiteres probabilistisches Verfahren diskutiert.

10.1 Randomized Rounding

Das Problem *0/1-ganzzahlige Programmierung* (oder Boolesche Optimierung) ist bekanntermaßen NP-vollständig. Hierbei seien $x_1, \ldots, x_n$ Variablen, die nur die Werte 0 und 1 annehmen dürfen. Die Aufgabe besteht darin, eine Wertebelegung für die Varia-

blen zu finden, die eine Linearkombination

$$\sum_{i=1}^{n} a_i x_i$$

mit gegebenen Koeffizienten $a_1, \ldots, a_n$ maximiert (oder alternativ: minimiert), unter Einhaltung von Nebenbedingungen, die in Form eines Systems von linearen Ungleichungen gegeben sind:

$$\begin{array}{rcl} \sum_{i=1}^{n} b_{1,i} x_i & \geq & c_1 \\ \sum_{i=1}^{n} b_{2,i} x_i & \geq & c_2 \\ & \vdots & \\ \sum_{i=1}^{n} b_{m,i} x_i & \geq & c_m \end{array}$$

Möglicherweise ist es – je nach Problemstellung – auch geschickt, in dem Ungleichungssystem noch weitere Hilfsvariablen (sog. *Schlupfvariablen*) z_j zu verwenden, deren Wertebelegung dann aber nicht Bestandteil der gesuchten Lösung ist.

Auf Grund der NP-Vollständigkeit dieses Problems kann also jedes NP-Problem in Form einer solchen 0/1-ganzzahligen Programmierungsaufgabe niedergeschrieben werden.

Da das Problem als solches wie gesagt NP-vollständig ist, haben wir keinen effizienten Algorithmus zur exakten Lösung anzubieten. Aber, es gibt eine interessante probabilistische Methode, die in vielen Fällen recht gute Näherungslösungen liefert. Diese Methode beruht auf der Tatsache, dass das sehr ähnlich aussehende Problem, bei dem die Variablen x_i beliebige *reelle* Werte in $[0, 1]$ annehmen dürfen, effizient lösbar ist. (Man spricht in diesem Zusammenhang von *Relaxation*; dies bedeutet ein gewisses „Aufweichen“ der Problemstellung, so dass effiziente Lösungsalgorithmen ermöglicht werden). Tatsächlich gibt es für diese modifizierte Problemstellung verschiedene effiziente Verfahren (z.B. den Algorithmus von Karmarkar), die wir hier nicht beschreiben wollen. (Ein recht einfach zu implementierendes Verfahren ist der Simplex-Algorithmus, der im „Durchschnittsfall“ sehr effizient arbeitet, der allerdings ein exponentielles worst-case Verhalten hat. Dieser Algorithmus fällt in die Kategorie der lokalen Verbesserungsstrategien, die wir Abschnitt 10.3 ansprechen).

Wir gehen also so vor, dass wir das Problem zunächst über $[0, 1]^n$ mit einem der existierenden Verfahren exakt lösen. Die Lösung $(a_1, \ldots, a_n) \in [0, 1]^n$ für die Variablen $(x_1, \ldots, x_n)$ wird im Allgemeinen aber nicht 0/1-wertig sein. Nun nehmen wir den Zufall zu Hilfe. Wir interpretieren jedes errechnete a_i als eine Wahrscheinlichkeit. Das heißt, wir führen dann n unabhängige Zufallsexperimente durch, indem wir zufällige reelle Zahlen zwischen 0 und 1 ziehen. Ist die i-te Zufallszahl $\leq a_i$, so setzen wir x_i auf 1; sonst setzen wir x_i auf 0 (also $Pr(x_i = 1) = a_i$). Dieser soeben beschriebene Rundungsvorgang hat den Namen *Randomized Rounding*. Hier nochmals als Algorithmus:

Algorithmus 10.1 `ramdomizedRounding`(a_i)

```
1: FOR i = 1 TO n DO
2:     Wähle eine reellwertige Zufallszahl z aus [0,1]
3:     IF z ≤ a_i THEN
4:         x_i = 1
5:     ELSE
6:         x_i = 0
7: output x_1 x_2 ... x_n
```

Nehmen wir an, $(a_1, a_2, \ldots, a_n) \in [0,1]^n$ sei die errechnete reellwertige Wertebelegung für die Variablen $(x_1, x_2, \ldots, x_n)$. Sei $(b_1, b_2, \ldots, b_n) \in \{0,1\}^n$ die tatsächlich gesuchte optimale Lösung. Je nach Problemstellung sollte der typische Abstand $|a_i - b_i|$ deutlich kleiner als 1/2 sein. Die Wahrscheinlichkeit, dass beim Randomized Rounding-Vorgang ein einzelner Wert für x_i korrekt (also gleich b_i) gesetzt wird, ist a_i, falls $b_i = 1$, und ist $1 - a_i$, falls $b_i = 0$. Anders ausgedrückt, die Wahrscheinlichkeit ist $1 - |a_i - b_i|$. Also wird insgesamt die korrekte Lösung $(b_1, \ldots, b_n)$ ermittelt mit Wahrscheinlichkeit $\prod_{i=1}^{n}(1 - |a_i - b_i|)$. Die Wahrscheinlichkeit, bei t unabhängigen Versuchen die Lösung nicht zu finden, ist dann

$$\left(1 - \prod_{i=1}^{n}(1 - |a_i - b_i|)\right)^t \leq e^{-t \cdot \prod_{i=1}^{n}(1-|a_i-b_i|)}$$

Das bedeutet, um eine Fehlerwahrscheinlichkeit von nur 2^{-30} zu erreichen, muss die Wiederholungszahl

$$t \geq \frac{30 \ln 2}{\prod_{i=1}^{n}(1 - |a_i - b_i|)}$$

gewählt werden. *Zahlenbeispiel:* Wenn $|a_i - b_i| \leq 0.3$ für alle i gilt, so ist $t = \mathcal{O}(1.4286^n)$.

10.2 Greedy-Heuristiken

In Kapitel 5 haben wir das Greedy Algorithmen-Paradigma vorgestellt, welches nach der Methode vorgeht, nur auf Grund der lokal verfügbaren Information den nächsten Lösungs-Erweiterungsschritt vorzunehmen, nämlich denjenigen, der für den Moment den größten Gewinn einbringt – bei einem Maximierungsproblem (bzw. die geringsten Kosten verursacht – bei einem Minimierungsproblem). Die Methode ist deshalb attraktiv, weil sie einfach zu implementieren und effizient ist.

Wenn die zugrundeliegende algebraische Struktur ein Matroid ist, so führt diese simple Strategie tatsächlich zum Erfolg, das heißt, es wird eine optimale Lösung gefunden. Wenn diese Voraussetzungen aber nicht gegeben sind, so wird Greedy im Allgemeinen nicht erfolgreich sein. Aber trotzdem, unter Umständen wird durch die simple Greedy-Vorgehensweise auch in einem solchen Fall eine „nahezu optimale“ Lösung erreicht. Daher haben wir es mit einem effizient implementierbaren *heuristischen* Verfahren zu tun.

Betrachten wir eine sehr einfache Greedy-Heuristik für das *Graphenfärbungsproblem*. (Gegeben ein Graph, färbe diesen mit möglichst wenigen Farben. Das Problem ist NP-vollständig). Wir ordnen nach irgendeinem Kriterium die Knoten des Graphen an. (Vernünftig ist es, die Knoten nach *absteigenden* Knotengraden zu ordnen). Dann geben wir dem ersten Knoten die Farbe 1; dem zweiten ebenso die Farbe 1, wenn er nicht mit dem ersten verbunden ist, ansonsten die Farbe 2. Der allgemeine Fall sieht so aus, dass wir immer die kleinstmögliche Farbennummer auswählen, bei der kein Konflikt mit den bisherigen Farben der Nachbarknoten entsteht. Man kann leicht Beispiele angeben, an denen man erkennt, dass das Verfahren nicht immer die bestmögliche Färbung findet (sonst gilt $P = NP$!).

Angewandt auf das *Traveling Salesman Problem* wäre das Folgende eine mögliche Realisierung einer Greedy-Heuristik-Idee: Konstruiere sukzessive eine Rundreise, indem man vom aktuellen Standort aus immer die *nächstliegende* Stadt, die vorher noch nicht besucht wurde, auswählt und besucht („Nearest Neighbor Heuristic“). Am Schluss kehrt man zur Ausgangsstadt zurück. (Wir gehen hier davon aus, dass von jeder Stadt aus zu jeder anderen eine mögliche Verbindung existiert).

Diese Heuristik liefert mäßige Ergebnisse. Die erzeugten Rundreisen enthalten zu Beginn zwar viele kurze Kanten, dann aber kommt es vor, dass eigentlich zu besuchende Knoten „übersehen“ werden, da sie von der aktuellen Position zu weit entfernt sind. Am Schluss aber müssen diese Knoten von einer noch weiter entfernten Position dann doch besucht werden, da keine unbesuchten Knoten mehr übrigbleiben.

Trotzdem können die mit einer solch einfachen Heuristik erzeugten Rundreisen nützlich sein, wenn man anschließend eine Methode der Nach-Verbesserung durchführt (siehe Abschnitt 10.3 und die danach folgenden Abschnitte). Die Komplexität des Verfahrens ist $\mathcal{O}(n^2)$, denn jeder der $n - 1$ Erweiterungsschritte erfordert eine Minimumssuche mit Komplexität $\mathcal{O}(n)$.

Kleverer ist dann schon folgende Greedy-Methode: Wir starten mit einer kurzen Tour, die nur über die zwei am dichtesten zueinander liegenden Knoten führt. In jedem Schritt wird diese Tour dann um einen weiteren Knoten erweitert, bis schließlich alle Knoten mit einbezogen sind. Welcher der noch nicht involvierten Knoten als nächster in die Tour integriert werden soll, das ist die Frage. Hier gibt es verschiedene Varianten, die untersucht wurden. Die Methode, die sich in praktischen Versuchen als die beste erwiesen hat, ist „Farthest Insertion“: Man wählt unter den noch nicht integrierten Knoten denjenigen, dessen Minimalabstand zu einem Knoten, der bereits auf der Tour liegt, *maximal* ist. Nicht viel schlechter verhält sich „Random Insertion“: Man wählt einen zufälligen, noch nicht integrierten Knoten. Der betreffende Knoten wird dann so in die Tour zwischen zwei Knoten eingefügt, dass die Zunahme der Tourenlänge minimal bleibt.

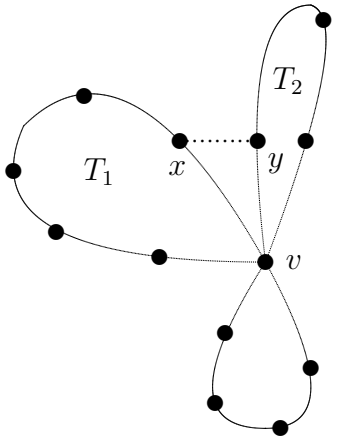
Es mag wundern, dass Farthest Insertion besser ist als das eigentlich offensichtlichere „Nearest Insertion“. Der Grund liegt intuitiv darin, dass Farthest Insertion schon nach relativ wenigen Schritten eine Tour erzeugt, die in der globalen Kontur der endgültigen Tour ähnelt.

Die Komplexität von Farthest Insertion und von Random Insertion ist $\mathcal{O}(n^2)$.

In der Praxis noch besser verhält sich folgende Greedy-Heuristik: Man wählt zunächst

einen beliebigen Knoten $v \in V$ als „Basisknoten“ aus. Zur Initialisierung des Verfahrens legt man von diesem Basisknoten aus sternförmig $n-1$ viele kurze Rundreisen $v-x-v$ zu jedem Knoten $x \in V - \{v\}$ an. Sukzessive werden nun in einer Schleife je zwei verschiedene, über v laufende Rundreisen T_1, T_2 ausgewählt, so dass $\{v, x\}$ eine Kante auf T_1 und $\{v, y\}$ eine Kante auf T_2 ist. Aus den beiden Touren T_1, T_2 wird nun eine einzige gemacht, indem man die Kanten $\{v, x\}, \{v, y\}$ streicht und statt dessen die Kante $\{x, y\}$ hinzunimmt.

Skizze:



Und zwar wählt man in jedem Schritt dasjenige Paar T_1, T_2 und deren Knoten x, y aus, welches die Kostenersparnis $w(\{v, x\}) + w(\{v, y\}) - w(\{x, y\})$ maximiert.

Implementieren lässt sich das Verfahren, indem man die Touren T_i als verkettete Listen darstellt. Jede Tour hat im Allgemeinen zwei Endpunkte – also Nachbarpunkte von v – die für den oben beschriebenen Vereinigungsvorgang in Frage kommen. Beim Einrichten einer Tour wird zugleich für die beiden Endpunkte der günstigste Endpunkt derjenigen Nachbartour vermerkt, mit welcher gegebenenfalls zu vereinigen ist. Der Vereinigungsvorgang erfordert das Zusammenfügen zweier Listen. Nach dem Vereinigungsvorgang erfolgt ein update bzgl. der günstigsten Nachbarn, welcher ungünstigenfalls die Komplexität $\mathcal{O}(n^2)$ hat. Daher können wir die Komplexität des Verfahrens insgesamt mit $\mathcal{O}(n^3)$ abschätzen.

10.3 Lokale Verbesserungsstrategien

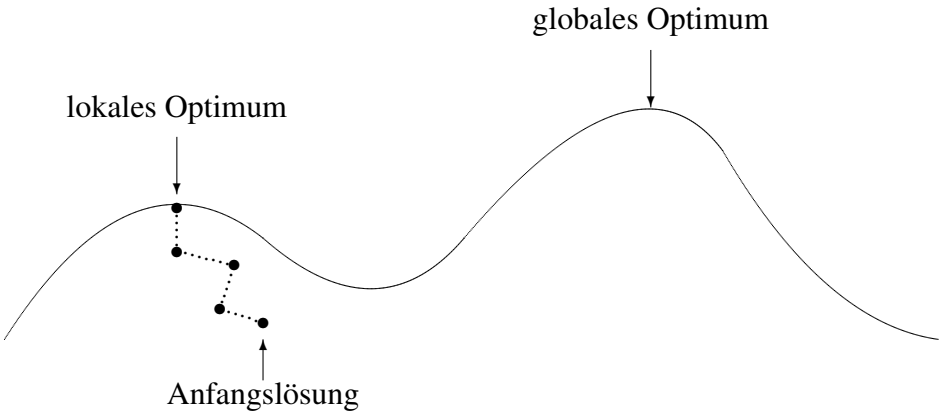
Die Idee besteht darin, mit einer beliebigen (zufälligen oder systematisch – evtl. mit einer Greedy Heuristik – gewählten) Lösung zu beginnen, und diese dann in einer zufälligen Weise durch gewisse lokale, geringfügige Veränderungen zu „mutieren“. Sollte diese Veränderung eine Verbesserung gebracht haben, so wird diese neue Lösung übernommen. Die Methode wird manchmal auch *hill climbing* oder *local search* genannt.

Der folgende Algorithmus skizziert dieses Vorgehen.

Algorithmus 10.2 `localSearch()`

1: Erzeuge eine Anfangslösung l
2: **REPEAT**
3: Modifiziere l zufällig zu l'
4: **IF** Lösung l' ist besser als l **THEN**
5: $l = l'$
6: **UNTIL** längere Zeit keine Verbesserung mehr eingetreten
7: Gib l als akzeptable Lösung aus

Das Problem bei diesem Vorgehen ist, dass man möglicherweise in einem lokalen Optimum hängenbleiben kann und das globale Optimum nicht findet. Das folgende Bild skizziert den Verlauf der Lösungsverbesserungen.



Dem Problem der lokalen Optima kann dadurch begegnet werden, dass man das Verfahren wiederholt ausführt mit immer wieder neuen zufälligen Anfangslösungen.

Man beachte, dass auch der Ford-Fulkerson Algorithmus (Abschnitt 6.5) in die Kategorie lokale Verbesserungsstrategie fällt. Allerdings wird in diesem Fall durch das Min-Cut-Max-Flow-Theorem (Seite 94) garantiert, dass es keine lokalen Optima gibt.

Es gibt einen recht erfolgreichen Algorithmus für das TSP, der nach dieser lokalen Verbesserungsmethode vorgeht. Gegeben sei dieses Mal eine *symmetrische* Entfernungsmatrix $M[1..n, 1..n]$. Außerdem nehmen wir (wie in Abschnitt 10.2) an, dass *alle* Kanten in dem zugrundeliegenden Graphen vorhanden sind. Dies hat die Konsequenz, dass *jede* Permutation eine zulässige Lösung ist.

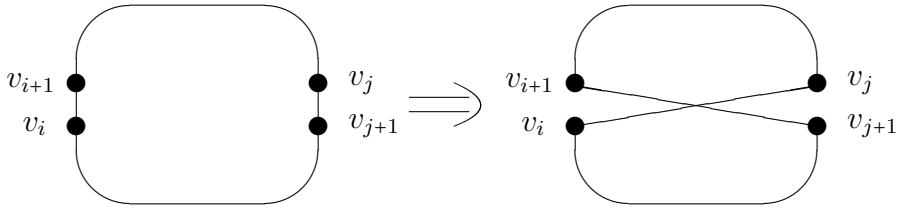
In der Initialisierungsphase wird eine zufällige Permutation aller Knoten erzeugt: $v_1, v_2, \ldots, v_n$. (Im Folgenden sei mit v_{n+1} wieder der Knoten v_1 gemeint).

Die lokalen Verbesserungsschritte vollziehen sich wie folgt („2-Opt-Heuristik“): Wähle zunächst zwei zufällige Knoten v_i und v_j. Diese dürfen auf der bisherigen Rundreise nicht direkt benachbart sein. Ferner sei (o.B.d.A.) $i < j$.

Falls jetzt

$$M[v_i, v_{i+1}] + M[v_j, v_{j+1}] \;>\; M[v_i, v_j] + M[v_{i+1}, v_{j+1}]$$

gilt, so kann die bisherige Rundreise verbessert werden. Das folgende Bild skizziert die Modifikation.

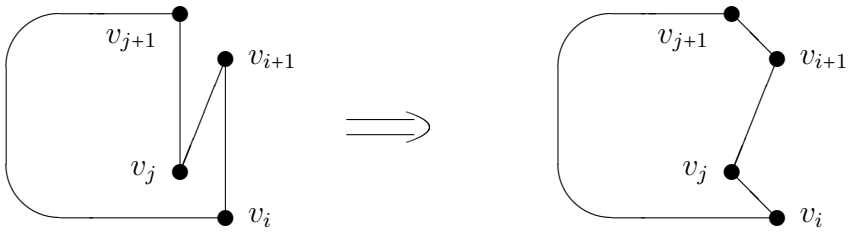


Mit anderen Worten, die bisherige Rundreise $v_1, \ldots, v_n$ wird modifiziert zu

$$v_1, v_2, \ldots v_i,\ v_j, v_{j-1}, \ldots, v_{i+2}, v_{i+1},\ v_{j+1}, \ldots, v_n$$

Man beachte, dass der Abschnitt $v_j \ldots v_{i+1}$ in der modifizierten Rundreise in umgekehrter Richtung als zuvor durchlaufen wird. Daher haben wir zu Beginn die Symmetrie der Entfernungsmatrix gefordert.

Geometrisch betrachtet bedeutet die 2-Opt-Heuristik gerade, dass eine überflüssige Überkreuzung zweier Wege auf der Rundreise „entknotet“ wird, was zu einer Verkürzung der Rundreise führt. Oder auch: es wird ein überflüssiger „Knick“ gegradigt, wie das folgende Beispiel zeigt:



Zur Komplexität dieser Greedy-Heuristik lässt sich relativ wenig sagen. Es ist nicht klar, ob ein lokales Optimum in polynomialer Zeit erreicht wird.

10.4 Simulated Annealing

Dem Problem, in einem lokalen Optimum „hängenzubleiben“, kann man auch mit der Methode *Simulated Annealing* (simuliertes Ausglühen oder Abkühlen) begegnen. Wie bei der Auskühlung eines Metalls oder bei der Kristallzüchtung geht man von einem flüssigen Zustand aus und überführt diesen langsam bis hin zu einem möglichst energiearmen, festen Zustand. Dabei werden evtl. auch zwischendurch Zustände höherer Energie eingenommen.

Die wesentliche Idee ist, dass mit gewisser abnehmender Wahrscheinlichkeit im Verlauf des Algorithmus auch wieder schlechtere Lösungen akzeptiert werden. Dies verhindert, dass der Algorithmus in lokalen Optima steckenbleibt. Ist nämlich mal eine lokal optimale Lösung erreicht, reicht unter Umständen die Palette der möglichen Mutationen nicht aus, dieses lokale Optimum wieder zu verlassen, wenn man nicht auch in Zwischenschritten wieder schlechtere Lösungen akzeptiert. Diese Idee geht auf Metropoplis et al. (1953) zurück. Man spricht in diesem Zusammenhang auch vom *Metropolis-Algorithmus*.

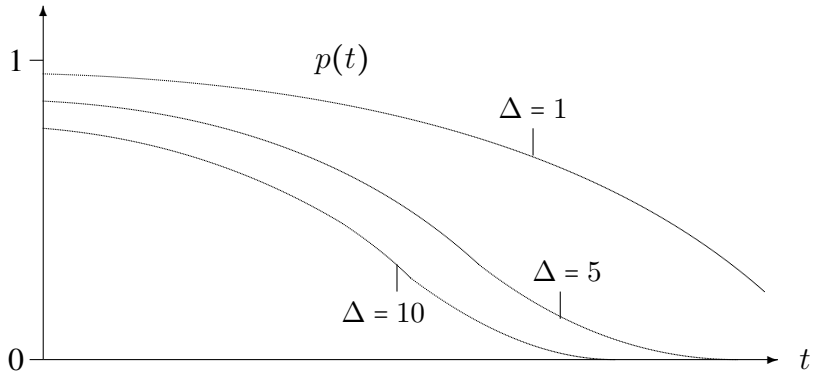
Der folgende Algorithmus skizziert das prinzipielle Vorgehen.

Algorithmus 10.3 `simulatedAnnealing()`

```
 1: Erzeuge eine Anfangslösung a
 2: Initialisiere c (die „Temperatur“) und l („Wiederholungsfaktor“)
 3: REPEAT
 4:    FOR k = 1 TO l DO
 5:       Erzeuge aus a eine zufällig mutierte Lösung a'
 6:       Δ := w(a') – w(a) //w ist die Gewichtsfunktion
 7:       IF Δ ≤ 0 THEN
 8:          a = a'
 9:       ELSE IF e^(-Δ/c) ≥ random THEN
10:          a = a'
11:    c = αc
12: UNTIL c < ε
```

Hierbei ist α eine Konstante kleiner als Eins, etwa $0.8 \leq \alpha \leq 0.99$, und ε ist eine kleine Konstante nahe bei Null. Eine Faustregel ist, den Startwert von c etwa 10-mal so groß wie den größtmöglichen, potenziellen Δ-Wert zu wählen. Ferner ist random eine gleichverteile Zufallszahl zwischen 0 und 1. Der Test $\Delta \leq 0$ bedeutet, dass die Lösung a' gegenüber a eine Verbesserung darstellt. (Die Formulierung hier ist für ein *Minimierungsproblem*, bei einem Maximierungsproblem müsste man Δ durch $-\Delta$ ersetzen).

Das folgende Diagramm skizziert, wie die Wahrscheinlichkeit $p = Pr[e^{-\Delta/c} \geq \mathit{random}] = e^{-\Delta/c}$ mit der Anzahl t der Schleifendurchläufe abnimmt. Hierbei ist zu berücksichtigen, dass $c = c(t) = c_0 \cdot \alpha^t$, und dementsprechend auch $p = p(t)$, eine Funktion von t ist.



Hier wurde die Abnahme-Geschwindigkeit für den Temperaturparameter $c = c(t) = c_0 \cdot \alpha^t$ gemäß einer geometrischen Reihe gewählt, so dass c also exponentiell gegen Null konvergiert. Wenn wir annehmen, dass der Anfangswert a für die Temperatur exponentiell in der Problemeingabelänge ist, so bedeutet dies, dass die Komplexität des Algorithmus polynomial ist. Andererseits besagt eine theoretische Betrachtung mit Hilfe der Theorie der Markoff-Ketten, dass eine Temperaturabnahme der Form $c(t) = \mathcal{O}(1/\log t)$ hinreichend

ist, um mit einer akzeptablen Wahrscheinlichkeit am Ende eine global-optimale Lösung zu erhalten. Setzt man eine solche Temperaturabnahme-Geschwindigkeit (auch „cooling schedule“ genannt) in obige Formeln ein, erhält man eine exponentielle Komplexität. Es gilt hier einen guten Kompromiss zwischen Komplexität und erwarteter Lösungsgüte zu schließen.

So wie hier formuliert, verwendet Simulated Annealing den Zufall, um bei schlechteren Lösungen festzulegen, ob diese beibehalten werden sollen oder nicht. Es spricht aber im Prinzip nichts dagegen, stattdessen diese Entscheidung deterministisch durchzuführen. Das bedeutet dann, dass der Algorithmus einen *Schwellenwert* s mitführt, so dass schlechtere Lösungen – ohne Verwenden von Zufall – dann akzeptiert werden, wenn $\Delta \leq s$. Dieser Schwellenwert (genauso wie im obigen Algorithmus die „Temperatur“ c) wird in jedem Schritt heruntergesetzt und strebt gegen Null. Diese Variante nennt man auch *Threshold Accepting*.

Ein weiteres technisches Problem bei Simulated Annealing ist, dass zwar gelegentlich ein Schritt akzeptiert wird, der die aktuelle Lösung verschlechtert, dass es aber im nächsten Schritt möglich ist, dass der vorangegangene Schritt wieder rückgängig gemacht wird. Deshalb wurde die sog. *Tabu-Suche* vorgeschlagen. Hierzu speichert man sich die letzten k erhaltenen Lösungen (für eine geeignete Konstante k) und verbietet alle Schritte, die auf eine der gespeicherten Lösungen zurückführt. Das heißt, es werden nur „neue“ Lösungen zugelassen.

10.5 Genetische Algorithmen

Genetische Algorithmen arbeiten vom Konzept her auch mit lokalen Verbesserungsstrategien, evtl. angereichert mit dem Simulated Annealing Konzept. Nur wird hier konsequenter das Vorbild der Natur, die genetische Evolution, imitiert.

Der erste Unterschied besteht darin, dass nicht nur eine einzelne Lösung verändert und verbessert wird, sondern eine *Population* von Lösungen. Lösungen werden in diesem Kontext auch oft *Chromosomen* genannt. Der weitere Unterschied besteht darin, dass nicht nur zufällige lokale Änderungen der Lösungen (Mutationen) vorgenommen werden, sondern auch *Kreuzungen* (auch *cross-over* oder *Rekombination* genannt) zwischen Lösungen, um solcherart neue Lösungen zu erzeugen. Nach dem Erzeugen solcher neuen Lösungen werden alle bewertet, und die neue Population besteht dann aus denjenigen mit den höchsten Bewertungen. (In diesem Kontext wird oft der Term „Fitness“ für die Bewertung verwendet). Zusammengefasst besteht ein genetischer Algorithmus aus den Komponenten Mutation, Rekombination und Selektion.

Im folgenden Algorithmus ist das Konzept ausgeführt.

Algorithmus 10.4 `genetic()`

1: Erzeuge eine zufällige Anfangspopulation von Lösungen $\{a_1, \ldots, a_m\}$
2: **REPEAT**
3: Erzeuge eine gewisse Anzahl zufälliger Mutationen der Lösungen
4: Erzeuge eine gewisse Anzahl zufälliger Kreuzungen von Lösungspaaren
5: Bewerte die Fitness aller erhaltenen Lösungen
6: Wähle die m fittesten Lösungen aus; diese seien dann $\{a_1, \ldots, a_m\}$
7: **UNTIL** Fitness verbessert sich nicht mehr
8: Finde in der erhaltenen Population die fitteste Lösung

Wir müssen noch diskutieren, wie die cross-over Operation zu verstehen ist. An dieser Stelle ist es sehr wichtig, dass die Lösungen „geeignet“ als Bitstrings codiert sind. Wir betrachten zwei Lösungen a_i und a_j und schneiden diese an einer beliebigen (oder geeigneten) Stelle (dem *cross-over point*) auf:

$$\begin{array}{rcl|l} a_i & = & 1110110111 & 1010101 \\ a_j & = & 0011011011 & 1110001 \end{array}$$

Nun sind zwei Kreuzungen möglich; man kann die vordere Hälfte von a_i (bzw. a_j) mit der hinteren Hälfte von a_j (bzw. a_i) kombinieren, und wenn die Codierung der Lösungen geeignet gewählt ist, so entstehen wieder zulässige Lösungen. Diese wären in diesem Fall

$$1110110111110001 \text{ und } 0011011011010101$$

Wichtig an der Codierung ist, dass die gekreuzten Lösungen in gewisser Weise die Charakteristika der „Eltern“ noch in sich tragen.

Eine andere Weise, zwei Eltern-Bitstrings miteinander zu kombinieren, um einen „gekreuzten“ Bitstring zu erhalten, stellt das *uniform cross-over* dar. Hierbei wird für $i = 1, 2, \ldots, n$ jeweils unabhängig mit Wahrscheinlichkeit 1/2 entschieden, ob das betreffende Bit des einen oder des anderen Elternteils zu übernehmen ist. Das hat die Konsequenz (wie bei der obigen Methode auch), dass Bitpositionen, in denen die Eltern identisch sind, im gekreuzten Bitstring übernommen werden.

Wenden wir uns dem TSP zu. Hier lässt sich das cross-over Konzept nicht in der angegebenen idealen, symmetrischen Form verwirklichen. Seien zwei Lösungen (also Rundreisen) gegeben:

$$v = (v_1, v_2, \ldots, v_n)$$
$$w = (w_1, w_2, \ldots, w_n)$$

Man kann nicht einfach eine vordere Hälfte der ersten Rundreise mit der hinteren Hälfte der zweiten Rundreise kombinieren, da dann Knoten (Städte) doppelt oder überhaupt nicht auftreten können.

Es gibt eine mögliche Lösung des Problems, die annehmbare Ergebnisse liefert: Wir wählen aus der ersten Lösung einen Teilabschnitt, den wir identisch übernehmen. Aus der zweiten Lösung schließen wir dann – ausgehend vom letzten besuchten Knoten des

ersten Teils – weitere Knoten an, sofern sie nicht schon vorgekommen sind; ansonsten lassen wir diese aus.

Beispiel:

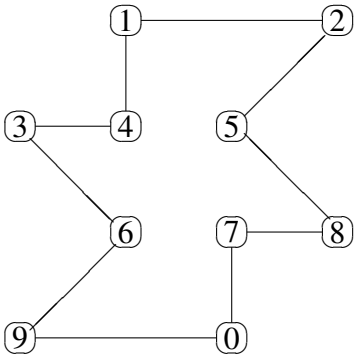
$$v = (5\ 6\ 2\ 0\ 1\ 4\ 7\ 3\ 8\ 9)$$

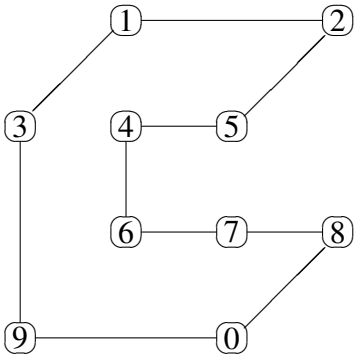
$$w = (3\ 2\ 1\ 6\ 5\ 9\ 4\ 0\ 8\ 7)$$

Wir wählen aus der ersten Lösung den Abschnitt $v' = (2\ 0\ 1\ 4\ 7\ 3)$. Wir setzen beim Nachfolgerknoten von 3 in w auf und wählen der Reihe nach diejenigen Knoten, die noch nicht vorgekommen sind. Das Ergebnis ist folgende „gekreuzte“ Rundreise:

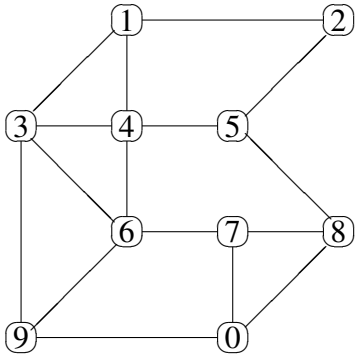
$$(2\ 0\ 1\ 4\ 7\ 3\ 6\ 5\ 9\ 8)$$

Eine andere Möglichkeit, den cross-over Operator beim (symmetrischen) Traveling Salesman Problem zu definieren, ist wie folgt. Wir erzeugen den Graphen, der sich ergibt, wenn man nur die in den beiden zu kreuzenden (Eltern-) Rundreisen verwendeten Kanten einträgt. Gegeben seien zum Beispiel die folgenden beiden Rundreisen:



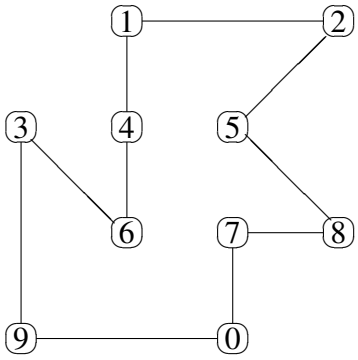


Wir tragen alle vorkommenden Kanten in einen gemeinsamen Graphen ein. Jeder Knoten in diesem Graphen hat jetzt 2,3 oder 4 Nachbarn.



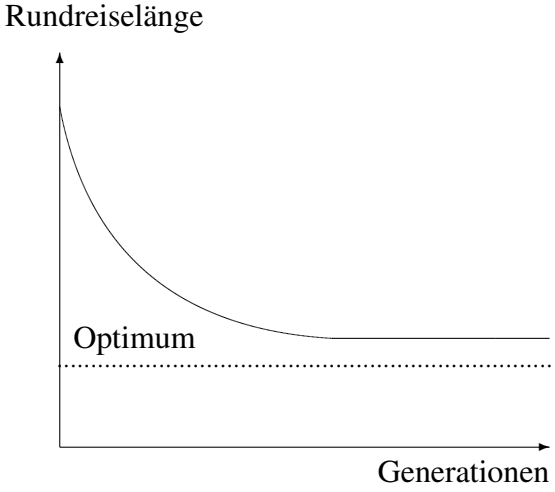
Nun versuchen wir probabilistisch durch einen „random walk“ auf dem Graphen, einen neue Rundreise zu finden. Hierbei kann es möglicherweise vorkommen, dass diese Zufallsirrfahrt in eine „Sackgasse“ gerät, also dass alle Nachbarn des aktuellen Knotens

auf der begonnenen Rundreise bereits besucht wurden. In einem solchen Fall muss eine Kante verwendet werden, die in keiner der Eltern-Rundreisen vorkam. Das folgende Bild zeigt eine Möglichkeit, die nur in den Eltern vorhandene Kanten verwendet:



Anstatt die Rundreise zufällig zu konstruieren, kann man auch eine Greedy-Heuristik verwenden, indem man immer denjenigen noch nicht besuchten Nachbarknoten mit minimaler (verbleibender) Kantenzahl auswählt.

Das folgende Diagramm illustriert, wie typischerweise im Laufe eines genetischen Algorithmus' die Lösungsgüte zunimmt (also die Länge der gefundenen Rundreisen abnimmt), wobei die optimale Lösung möglicherweise aber nicht erreicht wird.



Index