Vorlesungsskript

Theoretische Informatik III

Sommersemester 2008

Prof. Dr. Johannes Köbler
Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

30. Juli 2008

Inhaltsverzeichnis

1 Einleitung

In der VL ThI 2 standen folgende Themen im Vordergrund:

- Welche Probleme sind lösbar? (Berechenbarkeitstheorie)
- Welche Rechenmodelle sind adäquat? (Automatentheorie)
- Welcher Aufwand ist nötig? (Komplexitätstheorie)

Dagegen geht es in der VL ThI 3 in erster Linie um folgende Frage:

- Wie lassen sich eine Reihe von praktisch relevanten Problemstellungen möglichst effizient lösen? (Algorithmik)

Der Begriff *Algorithmus* geht auf den persischen Gelehrten Muhammed Al Chwarizmi (8./9. Jhd.) zurück. Der älteste bekannte nicht-triviale Algorithmus ist der nach *Euklid* benannte Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen (300 v. Chr.). Von einem Algorithmus wird erwartet, dass er jede *Problemeingabe* nach endlich vielen Rechenschritten löst (etwa durch Produktion einer Ausgabe). Ein Algorithmus ist ein „Verfahren“ zur Lösung eines Entscheidungs- oder Berechnungsproblems, das sich prinzipiell auf einer Turingmaschine implementieren lässt (Churchsche These bzw. Church-Turing-These).

Die Registermaschine

Bei Aussagen zur Laufzeit von Algorithmen beziehen wir uns auf die Registermaschine (engl. random access machine; RAM). Dieses Modell ist etwas flexibler als die Turingmaschine, da es den unmittelbaren Lese- und Schreibzugriff (random access) auf eine beliebige Speichereinheit (Register) erlaubt. Als Speicher stehen beliebig viele Register zur Verfügung, die jeweils eine beliebig große natürliche Zahl speichern können. Auf den Registerinhalten sind folgende arithmetische Operationen in einem Rechenschritt ausführbar: Addition, Subtraktion, abgerundetes Halbieren und Verdoppeln.

Die Laufzeit von RAM-Programmen wird wie bei TMs in der Länge der Eingabe gemessen. Man beachte, dass bei arithmetischen Problemen (wie etwa Multiplikation, Division, Primzahltests, etc.) die Länge einer Zahleingabe n durch die Anzahl $\lceil \log n \rceil$ der für die Binärkodierung von n benötigten Bits gemessen wird. Dagegen bestimmt bei nicht-arithmetischen Problemen (z.B. Graphalgorithmen oder Sortierproblemen) die Anzahl der gegebenen Zahlen die Länge der Eingabe.

Asymptotische Laufzeit und Landau-Notation

Definition 1. *Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$. Wir schreiben $f(n) = O(g(n))$, falls es Zahlen n_0 und c gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Die Bedeutung der Aussage $f(n) = O(g(n))$ ist, dass f „nicht wesentlich schneller“ als g wächst. Formal bezeichnet der Term $O(g(n))$ die Klasse aller Funktionen f, die obige Bedingung erfüllen. Die Gleichung $f(n) = O(g(n))$ drückt also in Wahrheit eine Element-Beziehung $f \in O(g(n))$ aus. O-Terme können auch auf der linken Seite vorkommen. In diesem Fall wird eine Inklusionsbeziehung ausgedrückt. So steht $n^2 + O(n) = O(n^2)$ für die Aussage $\{n^2 + f \mid f \in O(n)\} \subseteq O(n^2)$.

Beispiel 2.

- $7\log(n) + n^3 = O(n^3)$ *ist richtig.*
- $7\log(n)n^3 = O(n^3)$ *ist falsch.*

- $2^{n+O(1)} = O(2^n)$ *ist richtig.*
- $2^{O(n)} = O(2^n)$ *ist falsch (siehe Übungen).*

◁

Es gibt noch eine Reihe weiterer nützlicher Größenvergleiche von Funktionen.

Definition 3. *Wir schreiben $f(n) = o(g(n))$, falls es für jedes $c > 0$ eine Zahl n_0 gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Damit wird ausgedrückt, dass f „wesentlich langsamer" als g wächst. Außerdem schreiben wir

- *$f(n) = \Omega(g(n))$ für $g(n) = O(f(n))$, d.h. f wächst mindestens so schnell wie g)*
- *$f(n) = \omega(g(n))$ für $g(n) = o(f(n))$, d.h. f wächst wesentlich schneller als g, und*
- *$f(n) = \Theta(g(n))$ für $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$, d.h. f und g wachsen ungefähr gleich schnell.*

2 Suchen und Sortieren

2.1 Suchen von Mustern in Texten

In diesem Abschnitt betrachten wir folgende algorithmische Problemstellung.

String-Matching (StringMatching):

Gegeben: Ein Text $x = x_1 \cdots x_n$ und ein Muster $y = y_1 \cdots y_m$ über einem Alphabet Σ.

Gesucht: Alle Vorkommen von y in x.

Wir sagen *y kommt in x an Stelle i vor*, falls $x_{i+1} \cdots x_{i+m} = y$ ist. Typische Anwendungen finden sich in Textverarbeitungssystemen (emacs, grep, etc.), sowie bei der DNS- bzw. DNA-Sequenzanalyse.

Beispiel 4. *Sei $\Sigma = \{A, C, G, U\}$.*

$$\begin{aligned} \textit{Text } x &= \textit{AUGACGAUGAUGUAGGUAGCGUAGAUGAUGUAG}, \\ \textit{Muster } y &= \textit{AUGAUGUAG}. \end{aligned}$$

Das Muster y kommt im Text x an den Stellen 6 und 24 vor. ◁

Bei naiver Herangehensweise kommt man sofort auf folgenden Algorithmus.

Algorithmus `naiv-String-Matcher`(x, y)

```
1  Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2    V := ∅
3    for i := 0 to n − m do
4    if x_{i+1} ··· x_{i+m} = y_1 ··· y_m then
```

5 $V := V \cup \{i\}$
6 **Output:** V

Die Korrektheit von `naiv-String-Matcher` ergibt sich wie folgt:

- In der **for**-Schleife testet der Algorithmus alle potentiellen Stellen, an denen y in x vorkommen kann, und
- fügt in Zeile 4 genau die Stellen i zu V hinzu, für die $x_{i+1} \cdots x_{i+m} = y$ ist.

Die Laufzeit von `naiv-String-Matcher` lässt sich nun durch folgende Überlegungen abschätzen:

- Die **for**-Schleife wird $(n - m + 1)$-mal durchlaufen.
- Der Test in Zeile 4 benötigt maximal m Vergleiche.

Dies führt auf eine Laufzeit von $O(nm) = O(n^2)$. Für Eingaben der Form $x = a^n$ und $y = a^{\lfloor n/2 \rfloor}$ ist die Laufzeit tatsächlich $\Theta(n^2)$.

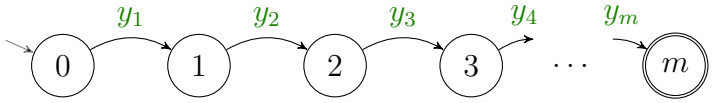
2.1.1 String-Matching mit endlichen Automaten

Durch die Verwendung eines endlichen Automaten lässt sich eine erhebliche Effizienzsteigerung erreichen. Hierzu konstruieren wir einen DFA M_y, der jedes Vorkommen von y in der Eingabe x durch Erreichen eines Endzustands anzeigt. M_y erkennt also die Sprache

$$L = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}.$$

Konkret konstruieren wir $M_y = (Z, \Sigma, \delta, 0, m)$ wie folgt:

- M_y hat $m + 1$ Zustände, die den $m + 1$ Präfixen $y_1 \cdots y_k$, $k = 0, \ldots, m$, von y entsprechen, d.h. $Z = \{0, \ldots, m\}$.
- Liest M_y im Zustand k das Zeichen y_{k+1}, so wechselt M_y in den Zustand $k + 1$, d.h. $\delta(k, y_{k+1}) = k + 1$ für $k = 0, \ldots, m - 1$:



- Falls das nächste Zeichen a nicht mit y_{k+1} übereinstimmt (engl. *mismatch*), wechselt M_y in den Zustand

$$\delta(k, a) = \max\{j \leq m \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}.$$

M_y speichert also in seinem Zustand die maximale Präfixlänge k, für die $y_1 \cdots y_k$ ein Suffix der gelesenen Eingabe ist:

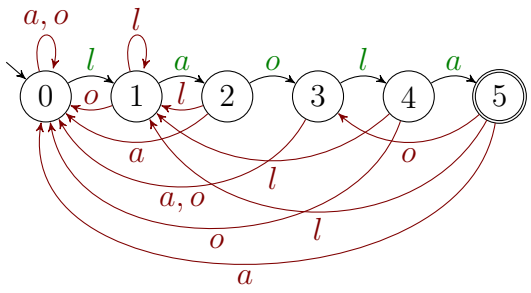
$$\hat{\delta}(0, x) = \max\{k \leq m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}.$$

Die Korrektheit von M_y folgt aus der Beobachtung, dass M_y isomorph zum *Äquivalenzklassenautomaten* M_{R_L} für L ist. M_{R_L} hat die Zustände $[y_1 \cdots y_k]$, $k = 0, \ldots, m$, von denen nur $[y_1 \cdots y_m]$ ein Endzustand ist. Die Überführungsfunktion ist definiert durch

$$\delta([y_1 \cdots y_k], a) = [y_1 \cdots y_j],$$

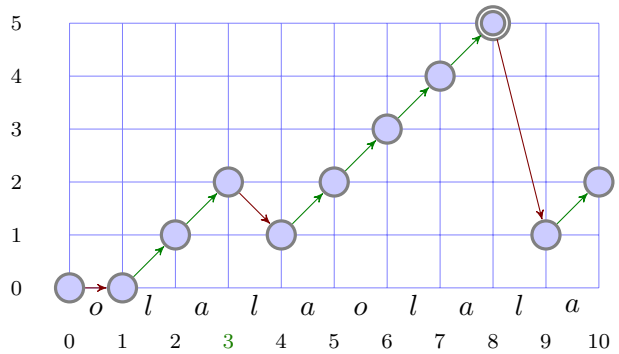
wobei $y_1 \cdots y_j$ das längste Präfix von $y = y_1 \cdots y_m$ ist, welches Suffix von $y_1 \cdots y_j a$ ist (siehe Übungen).

Beispiel 5. *Für das Muster $y = laola$ hat M_y folgende Gestalt:*



δ	0	1	2	3	4	5
a	0	2	0	0	5	0
l	1	1	1	4	1	1
o	0	0	3	0	0	3

M_y macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



◁

Insgesamt erhalten wir somit folgenden Algorithmus.

Algorithmus `DFA-String-Matcher`(x, y)

```
1  Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2    konstruiere den DFA M_y = (Z, Σ, δ, 0, m)
3    V := ∅
4    k := 0
5    for i := 1 to n do
6      k := δ(k, x_i)
7      if k = m then V := V ∪ {i − m}
8  Output: V
```

Die Korrektheit von `DFA-String-Matcher` ergibt sich unmittelbar aus der Tatsache, dass M_y die Sprache

$$L(M_y) = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}$$

erkennt. Folglich fügt der Algorithmus genau die Stellen $j = i - m$ zu V hinzu, für die y ein Suffix von $x_1 \cdots x_i$ (also $x_{j+1} \cdots x_{j+m} = y$) ist.

Die Laufzeit von `DFA-String-Matcher` ist die Summe der Laufzeiten für die Konstruktion von M_y und für die Simulation von M_y bei Eingabe x, wobei letztere durch $O(n)$ beschränkt ist. Für δ ist eine Tabelle mit $(m+1)\|\Sigma\|$ Einträgen

$$\delta(k, a) = \max\{j \leq k+1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}$$

zu berechnen. Jeder Eintrag $\delta(k, a)$ ist in Zeit $O(k^2) = O(m^2)$ berechenbar. Dies führt auf eine Laufzeit von $O(\|\Sigma\|m^3)$ für die Konstruktion von M_y und somit auf eine Gesamtlaufzeit von $O(\|\Sigma\|m^3 + n)$. Tatsächlich lässt sich M_y sogar in Zeit $O(\|\Sigma\|m)$ konstruieren.

2.1.2 Der Knuth-Morris-Pratt-Algorithmus

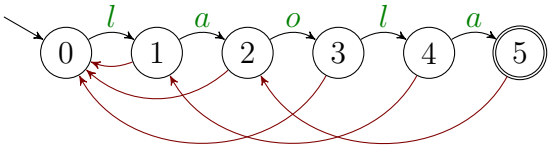
Durch eine Modifikation des Rücksprungmechanismus' lässt sich die Laufzeit von `DFA-String-Matcher` auf $O(n+m)$ verbessern. Hierz vergegenwärtigen wir uns folgende Punkte:

- Tritt im Zustand k ein Mismatch $a \neq y_{k+1}$ auf, so ermittelt M_y das längste Präfix p von $y_1 \cdots y_k$, das zugleich Suffix von $y_1 \cdots y_k a$ ist, und springt in den Zustand $k' = |p|$.
- Im Fall $k' \neq 0$ hat p also die Form $p = p'a$, wobei p' sowohl echtes Präfix als auch echtes Suffix von $y_1 \cdots y_k$ ist.
- Die Idee beim KMP-Algorithmus ist nun, unabhängig von a auf das nächst kleinere Präfix $\tilde{p}$ von $y_1 \cdots y_k$ zu springen, das auch Suffix von $y_1 \cdots y_k$ ist.
- Dies wird solange wiederholt, bis das Zeichen a „passt" (also $\tilde{p}a$ Präfix von y ist) oder der Zustand 0 erreicht wird.

Der KMP-Algorithmus besucht also alle Zustände, die auch M_y besucht, führt aber die Rücksprünge in mehreren Etappen aus. Die Sprungadressen werden durch die so genannte *Präfixfunktion* $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ ermittelt:

$$\pi(k) = \max\{0 \leq j \leq k-1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k\}.$$

Beispiel 6. *Für das Muster $y = laola$ ergibt sich folgende Präfixfunktion π:*

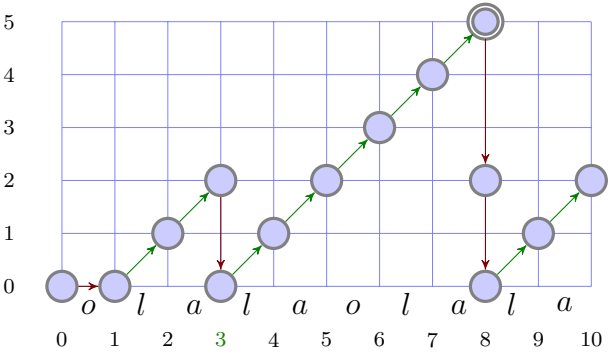


k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

Wir können uns die Arbeitsweise dieses Automaten wie folgt vorstellen:

1. *Erlaubt das nächste Eingabezeichen einen Übergang vom aktuellen Zustand k nach $k+1$, so führe diesen aus.*
2. *Ist ein Übergang nach $k+1$ nicht möglich und $k \geq 1$, so springe in den Zustand $\pi(k)$ ohne das nächste Zeichen zu lesen.*
3. *Andernfalls (d.h. $k = 0$ und ein Übergang nach 1 ist nicht möglich) lies das nächste Zeichen und bleibe im Zustand 0.*

Der KMP-Algorithmus macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



◁

Auf die Frage, wie sich die Präfixfunktion π möglichst effizient berechnen lässt, werden wir später zu sprechen kommen. Wir betrachten zunächst das Kernstück des KMP-Algorithmus, das sich durch eine leichte Modifikation von `DFA-String-Matcher` ergibt.

`DFA-String-Matcher`(x, y)

```
1  Input: Text x_1 ··· x_n
          und Muster y_1 ··· y_m
2  konstruiere M_y
3  V := ∅
4  k := 0
5  for i := 1 to n do
6    k := δ(k, x_i)
7    if k = m then
8      V := V ∪ {i − m}
9  Output: V
```

`KMP-String-Matcher`(x, y)

```
1  Input: Text x_1 ··· x_n und
          Muster y_1 ··· y_m
2  π := KMP-Prefix(y)
3  V := ∅
4  k := 0
5  for i := 1 to n do
6    while (k > 0 ∧ x_i ≠ y_{k+1}) do
7      k := π(k)
8    if x_i = y_{k+1} then k := k + 1
9    if k = m then
10     V := V ∪ {i − m}, k := π(k)
11 Output: V
```

Die Korrektheit des Algorithmus `KMP-String-Matcher` ergibt sich einfach daraus, dass er den Zustand m an genau den gleichen Textstellen besucht wie `DFA-String-Matcher`, und somit wie dieser alle Vorkommen von y im Text x findet.

Für die Laufzeitanalyse von `KMP-String-Matcher` (ohne die Berechnung von `KMP-Prefix`) stellen wir folgende Überlegungen an.

- Die Laufzeit ist proportional zur Anzahl der Zustandsübergänge.
- Bei jedem Schritt wird der Zustand um maximal Eins erhöht.
- Daher kann der Zustand nicht öfter verkleinert werden als er erhöht wird (*Amortisationsanalyse*).
- Es gibt genau n Zustandsübergänge, bei denen der Zustand erhöht wird bzw. unverändert bleibt.
- Insgesamt finden also höchstens $2n = O(n)$ Zustandsübergänge statt.

Nun kommen wir auf die Frage zurück, wie sich die Präfixfunktion π effizient berechnen lässt. Die Aufgabe besteht darin, für jedes Präfix

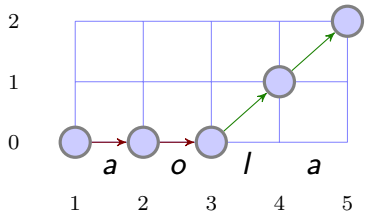
$y_1 \cdots y_i$, $i \geq 1$, das längste echte Präfix zu berechnen, das zugleich Suffix von $y_1 \cdots y_i$ ist.

Die Idee besteht nun darin, mit dem KMP-Algorithmus das Muster y im Text $y_2 \cdots y_m$ zu suchen. Dann liefert der beim Lesen von y_i erreichte Zustand k gerade das längste Präfix $y_1 \cdots y_k$, das zugleich Suffix von $y_2 \cdots y_i$ ist (d.h. es gilt $\pi(i) = k$). Zudem werden bis zum Lesen von y_i nur Zustände kleiner als i erreicht. Daher sind die π-Werte für alle bis dahin auszuführenden Rücksprünge bereits bekannt und π kann in Zeit $O(m)$ berechnet werden.

Prozedur `KMP-Prefix`(y)

```
1  π(1) := 0
2  k := 0
3  for i := 2 to m do
4     while (k > 0 ∧ y_i ≠ y_{k+1}) do k := π(k)
5     if y_i = y_{k+1} then k := k + 1
6     π(i) := k
7  return(π)
```

Beispiel 7. *Die Verarbeitung des Musters y = laola durch* `KMP-Prefix` *ergibt folgendes Ablaufprotokoll:*



k	1	2	3	4	5
$\pi(k)$	0	0	0	1	2

◁

Wir fassen die Laufzeiten der in diesem Abschnitt betrachteten String-Matching Algorithmen in einer Tabelle zusammen:

Algorithmus	Vorverarbeitung	Suche	Gesamtlaufzeit
naiv	0	$O(nm)$	$O(nm)$
DFA (einfach)	$O(\|\Sigma\|m^3)$	$O(n)$	$O(\|\Sigma\|m^3 + n)$
DFA (verbessert)	$O(\|\Sigma\|m)$	$O(n)$	$O(\|\Sigma\|m + n)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$	$O(n)$

2.2 Durchsuchen von Zahlenfolgen

Als nächstes betrachten wir folgendes Suchproblem.

Element-Suche

Gegeben: Eine Folge $a_1, \ldots, a_n$ von natürlichen Zahlen und eine Zahl a.

Gesucht: Ein Index i mit $a_i = a$ (bzw. eine Fehlermeldung, falls $a \notin \{a_1, \ldots, a_n\}$ ist).

Typische Anwendungen finden sich bei der Verwaltung von Datensätzen, wobei jeder Datensatz über einen eindeutigen Schlüssel (z.B. *Matrikelnummer*) zugreifbar ist. Bei manchen Anwendungen können die Zahlen in der Folge auch mehrfach vorkommen. Gesucht sind dann evtl. alle Indizes i mit $a_i = a$. Durch eine sequentielle Suche lässt sich das Problem in Zeit $O(n)$ lösen.

Algorithmus `Sequential-Search`

```
1  Input: Eine Zahlenfolge a_1, ..., a_n und eine Zahl a
2     i := 0
3     repeat
4        i := i + 1
5     until (i = n ∨ a = a_i)
6  Output: i, falls a_i = a bzw. Fehlermeldung, falls
       a_i ≠ a
```

Falls die Folge $a_1, \ldots, a_n$ sortiert ist, d.h. es gilt $a_i \leq a_j$ für $i \leq j$, bietet sich eine *Binärsuche* an.

Algorithmus `Binary-Search`

```
1  Input: Eine Zahlenfolge a_1, ..., a_n und eine Zahl a
2    l := 1
3    r := n
4    while l < r do
5      m := ⌊(l + r)/2⌋
6      if a ≤ a_m then r := m else l := m + 1
7  Output: l, falls a_l = a bzw. Fehlermeldung, falls
       a_l ≠ a
```

Offensichtlich gibt der Algorithmus im Fall $a \notin \{a_1, \ldots, a_n\}$ eine Fehlermeldung aus. Im Fall $a \in \{a_1, \ldots, a_n\}$ gilt die *Schleifeninvariante* $a_l \leq a \leq a_r$. Daher muss nach Abbruch der **while**-Schleife $a = a_l$ sein. Dies zeigt die Korrektheit von `Binary-Search`.

Da zudem die Länge $l - r + 1$ des Suchintervalls $[l, r]$ in jedem Schleifendurchlauf mindestens auf $\lfloor (l - r)/2 \rfloor + 1$ reduziert wird, werden höchstens $\lceil \log n \rceil$ Schleifendurchläufe ausgeführt. Folglich ist die Laufzeit von `Binary-Search` höchstens $O(\log n)$.

2.3 Sortieralgorithmen

Wie wir im letzten Abschnitt gesehen haben, lassen sich Elemente in einer sortierten Folge sehr schnell aufspüren. Falls wir diese Operation öfters ausführen müssen, bietet es sich an, die Zahlenfolge zu sortieren.

Sortierproblem

Gegeben: Eine Folge $a_1, \ldots, a_n$ von natürlichen Zahlen.

Gesucht: Eine Permutation $a_{i_1}, \ldots, a_{i_n}$ dieser Folge mit $a_{i_j} \leq a_{i_{j+1}}$ für $j = 1, \ldots, n - 1$.

Man unterscheidet *vergleichende Sortierverfahren* von den übrigen Sortierverfahren. Während erstere nur Ja-Nein-Fragen der Form „$a_i \leq a_j$?“ oder „$a_i < a_j$?“ stellen dürfen, können letztere auch die konkreten Zahlenwerte a_i der Folge abfragen. Vergleichsbasierte Verfahren benötigen im schlechtesten Fall $\Omega(n \log n)$ Vergleiche, während letztere unter bestimmten Zusatzvoraussetzungen sogar in Linearzeit arbeiten.

2.3.1 Sortieren durch Einfügen

Ein einfacher Ansatz, eine Zahlenfolge zu sortieren, besteht darin, sequentiell die Zahl a_i $(i = 2, \ldots, n)$ in die bereits sortierte Teilfolge $a_1, \ldots, a_{i-1}$ einzufügen.

Algorithmus `Insertion-Sort`$(a_1, \ldots, a_n)$

```
1    for i := 2 to n do z := a_i
2      j := i − 1
3      while (j ≥ 1 ∧ a_j > z) do
4        a_{j+1} := a_j
5        j := j − 1
6      a_{j+1} := z
```

Die Korrektheit von `Insertion-Sort` lässt sich induktiv durch den Nachweis folgender Schleifeninvarianten beweisen:

- Nach jedem Durchlauf der **for**-Schleife sind $a_1, \ldots, a_i$ sortiert.
- Nach jedem Durchlauf der **while**-Schleife gilt $z < a_k$ für $k = j + 2, \ldots, i$.

Zusammen mit der Abbruchbedingung der **while**-Schleife folgt hieraus, dass z in Zeile 5 an der jeweils richtigen Stelle eingefügt wird.

Da zudem die **while**-Schleife für jedes $i = 2, \ldots, n$ höchstens $(i - 1)$-mal ausgeführt wird, ist die Laufzeit von `Insertion-Sort` durch $\sum_{i=2}^{n} O(i - 1) = O(n^2)$ begrenzt.

Bemerkung 8.

- *Ist die Eingabefolge $a_1, \ldots, a_n$ bereits sortiert, so wird die* **while**-*Schleife niemals durchlaufen. Im* besten Fall *ist die Laufzeit daher* $\sum_{i=2}^{n} \Theta(1) = \Theta(n)$.
- *Ist die Eingabefolge $a_1, \ldots, a_n$ dagegen absteigend sortiert, so wandert z in $i-1$ Durchläufen der* **while**-*Schleife vom Ende an den Anfang der bereits sortierten Teilfolge $a_1, \ldots, a_i$. Im* schlechtesten Fall *ist die Laufzeit also* $\sum_{i=2}^{n} \Theta(i-1) = \Theta(n^2)$.
- *Bei einer zufälligen Eingabepermutation der Folge $1, \ldots, n$ wird z im Erwartungswert in der Mitte der Teilfolge $a_1, \ldots, a_i$ eingefügt. Folglich beträgt die (erwartete) Laufzeit im* durchschnittlichen Fall *ebenfalls* $\sum_{i=2}^{n} \Theta(\frac{i-1}{2}) = \Theta(n^2)$.

2.3.2 Sortieren durch Mischen

Wir können eine Zahlenfolge auch sortieren, indem wir sie in zwei Teilfolgen zerlegen, diese durch rekursive Aufrufe sortieren und die sortierten Teilfolgen wieder zu einer Liste zusammenfügen.

Diese Vorgehensweise ist unter dem Schlagwort "Divide and Conquer" (auch "divide et impera", also "teile und herrsche") bekannt. Dabei wird ein Problem gelöst, indem man es

- in mehrere Teilprobleme aufteilt,
- die Teilprobleme rekursiv löst, und
- die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammenfügt.

Die Prozedur `Mergesort`(A, l, r) sortiert ein Feld $A[l \ldots r]$, indem sie

- es in die Felder $A[l \ldots m]$ und $A[m+1 \ldots r]$ zerlegt,
- diese durch jeweils einen rekursiven Aufruf sortiert, und
- die sortierten Teilfolgen durch einen Aufruf der Prozedur `Merge`(A, l, m, r) zu einer sortierten Folge zusammenfügt.

Algorithmus `Mergesort`(A, l, r)

```
1  if l < r then
2     m := ⌊(l + r)/2⌋
3     Mergesort(A, l, m)
4     Mergesort(A, m + 1, r)
5     Merge(A, l, m, r)
```

Die Prozedur `Merge`(A, l, m, r) mischt die beiden sortierten Felder $A[l \ldots m]$ und $A[m+1 \ldots r]$ zu einem sortierten Feld $A[l \ldots r]$.

Prozedur `Merge`(A, l, m, r)

```
1  allokiere Speicher fuer ein neues Feld B[l ... r]
2  j := l
3  k := m + 1
4  for i := l to r do
5     if j > m then
6        B[i] := A[k]
7        k := k + 1
8     else if k > r then
9        B[i] := A[j]
10       j := j + 1
11    else if A[j] ≤ A[k] then
12       B[i] := A[j]
13       j := j + 1
14    else
15       B[i] := A[k]
16       k := k + 1
17 kopiere das Feld B[l ... r] in das Feld A[l ... r]
18 gib den Speicher fuer B wieder frei
```

Man beachte, dass `Merge` für die Zwischenspeicherung der gemischten Folge zusätzlichen Speicher benötigt. `Mergesort` ist daher kein *"in place"-Sortierverfahren*, welches neben dem Speicherplatz für die Eingabefolge nur konstant viel zusätzlichen Speicher belegen darf.

Zum Beispiel ist `Insertion-Sort` ein "in place"-Verfahren. Auch `Mergesort` kann als ein "in place"-Sortierverfahren implementiert werden, falls die zu sortierende Zahlenfolge nicht als Array, sondern als mit Zeigern verkettete Liste vorliegt (hierzu muss allerdings auch noch die Rekursion durch eine Schleife ersetzt werden).

Unter der Voraussetzung, dass `Merge` korrekt arbeitet, können wir per Induktion über die Länge $n = r - l + 1$ des zu sortierenden Arrays die Korrektheit von `Mergesort` wie folgt beweisen:

$n = 1$**:** In diesem Fall tut `Mergesort` nichts, was offensichtlich korrekt ist.

$n \rightsquigarrow n + 1$**:** Um eine Folge der Länge $n + 1 \geq 2$ zu sortieren, zerlegt sie `Mergesort` in zwei Folgen der Länge höchstens n. Diese werden durch die rekursiven Aufrufe nach IV korrekt sortiert und von `Merge` nach Voraussetzung korrekt zusammengefügt.

Die Korrektheit von `Merge` lässt sich leicht induktiv durch den Nachweis folgender Invariante für die FOR-Schleife beweisen:

- Nach jedem Durchlauf enthält $B[l \cdots i]$ die $i - l + 1$ kleinsten Elemente aus $A[l \cdots m]$ und $A[m + 1 \cdots r]$ in sortierter Reihenfolge.
- Hierzu wurden die ersten $j - 1$ Elemente von $A[l \cdots m]$ und die ersten $k - 1$ Elemente von $A[m + 1 \cdots r]$ nach B kopiert.

Nach dem letzten Durchlauf (d.h. $i = r$) enthält daher $B[l \cdots r]$ alle $r - l + 1$ Elemente aus $A[l \cdots m]$ und $A[m + 1 \cdots r]$ in sortierter Reihenfolge, womit die Korrektheit von `Merge` bewiesen ist.

Um eine Schranke für die Laufzeit von `Mergesort` zu erhalten, schätzen wir zunächst die Anzahl $V(n)$ der Vergleiche ab, die `Mergesort` (im schlechtesten Fall) benötigt, um ein Feld $A[l \cdots m]$ der Länge $n = r - l + 1$ zu sortieren. Wir bezeichnen die Anzahl der Vergleiche, die `Merge` benötigt, um die beiden sortierten Felder $A[l \ldots m]$ und $A[m + 1 \ldots r]$ zu mischen, mit $M(n)$. Dann erfüllt $V(n)$ die Rekursionsgleichung

$$V(n) = \begin{cases} 0, & \text{falls } n = 1, \\ V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + M(n), & n \geq 2. \end{cases}$$

Offensichtlich benötigt `Merge` $M(n) = n - 1$ Vergleiche. Falls n also eine Zweierpotenz ist, genügt es, folgende Rekursion zu lösen:

$$V(1) = 0 \text{ und } V(n) = 2V(n/2) + n - 1, n \geq 2.$$

Hierzu betrachten wir die Funktion $f(k) = V(2^k)$. Dann gilt

$$f(0) = 0 \text{ und } f(k) = 2f(k-1) + 2^k - 1, k \geq 1.$$

Aus den ersten Folgengliedern

$$\begin{array}{lcll} f(0) & = & 0, & \\ f(1) & = & 1, & \\ f(2) & = & 2 + 2^2 - 1 & = 1 \cdot 2^2 + 1, \\ f(3) & = & 2 \cdot 2^2 + 2 + 2^3 - 1 & = 2 \cdot 2^3 + 1, \\ f(4) & = & 2 \cdot 2 \cdot 2^3 + 2 + 2^4 - 1 & = 3 \cdot 2^4 + 1 \end{array}$$

lässt sich vermuten, dass $f(k) = (k - 1) \cdot 2^k + 1$ ist. Dies lässt sich leicht durch Induktion über k verifizieren, so dass wir für V die Lösungsfunktion $V(n) = n \log_2 n - n + 1$ erhalten.

Wir fassen unsere Beobachtungen zur Komplexität von `MergeSort` zusammen: Falls n eine Zweierpotenz ist, stellt `MergeSort` im schlechtesten Fall $V(n) = n \log_2 n - n + 1$ Fragen. Ist n keine Zweierpotenz, so können wir die Anzahl der Fragen durch $V(n') \leq V(2n) = O(V(n))$ abschätzen, wobei $n' \leq 2n$ die kleinste Zweierpotenz größer als n ist. Zudem ist leicht zu sehen, dass die Laufzeit $T(n)$ von `MergeSort` asymptotisch durch die Anzahl $V(n)$ der Vergleiche beschränkt ist, d.h. es gilt $T(n) = O(V(n))$.

Satz 9. *`MergeSort` ist ein vergleichendes Sortierverfahren mit einer Laufzeit von $O(n \log n)$.*

2.3.3 Lösen von Rekursionsgleichungen

Im Allgemeinen liefert der "Divide and Conquer"-Ansatz einfach zu implementierende Algorithmen mit einfachen Korrektheitsbeweisen. Die Laufzeit $T(n)$ erfüllt dann eine Rekursionsgleichung der Form

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n \text{ „klein“ ist,} \\ D(n) + \sum_{i=1}^{\ell} T(n_i) + C(n), & \text{sonst.} \end{cases}$$

Dabei ist $D(n)$ der Aufwand für das Aufteilen der Probleminstanz und $C(n)$ der Aufwand für das Verbinden der Teillösungen. Um solche Rekursionsgleichungen zu lösen, kann man oft eine Lösung „raten“ und per Induktion beweisen. Mit Hilfe von Rekursionsbäumen lassen sich Lösungen auch „gezielt raten“. Eine asymptotische Abschätzung liefert folgender Hauptsatz der Laufzeitfunktionen (Satz von Akra & Bazzi).

Satz 10 (Mastertheorem)**.** *Sei $T : \mathbb{N} \to \mathbb{N}$ eine Funktion der Form*

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + f(n),$$

mit $\ell, n_1, \ldots, n_\ell \in \mathbb{N}$ und $n_i \in \{\lfloor \alpha_i n \rfloor, \lceil \alpha_i n \rceil\}$ für fest gewählte reelle Zahlen $0 < \alpha_i < 1$ für $i = 1, \ldots, \ell$. Dann gilt im Fall $f(n) = \Theta(n^k)$ mit $k \geq 0$:

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

wobei c Lösung der Gleichung $\sum_{i=1}^{\ell} \alpha_i^c = 1$ ist.

2.3.4 Eine untere Schranke für das Sortierproblem

Frage. *Wie viele Vergleichsfragen benötigt ein vergleichender Sortieralgorithmus A mindestens, um eine Folge $(a_1, \ldots, a_n)$ von n Zahlen zu sortieren?*

Zur Beantwortung dieser Frage betrachten wir alle $n!$ Eingabefolgen $(a_1, \ldots, a_n)$ der Form $(\pi(1), \ldots, \pi(n))$, wobei $\pi \in S_n$ eine beliebige Permutation auf der Menge $\{1, \ldots, n\}$ ist. Um diese Folgen korrekt zu sortieren, muss A solange Fragen der Form $a_i < a_j$ (bzw. $\pi(i) < \pi(j)$) stellen, bis höchstens noch eine Permutation $\pi \in S_n$ mit den erhaltenen Antworten konsistent ist. Damit A möglichst viele Fragen stellen muss, beantworten wir diese so, dass mindestens die Hälfte der verbliebenen Permutationen mit unserer Antwort konsistent ist (*Mehrheitsvotum*). Diese Antwortstrategie stellt sicher, dass nach i Fragen noch mindestens $n!/2^i$ konsistente Permutationen übrig bleiben. Daher muss A mindestens

$$\lceil \log_2(n!) \rceil = n \log_2 n - n \log_2 e + \Theta(\log n) = n \log_2 n - \Theta(n)$$

Fragen stellen, um die Anzahl der konsistenten Permutationen auf Eins zu reduzieren.

Satz 11. *Ein vergleichendes Sortierverfahren benötigt mindestens $\lceil \log_2(n!) \rceil$ Fragen, um eine Folge $(a_1, \ldots, a_n)$ von n Zahlen zu sortieren.*

Wir können das Verhalten von A auch durch einen Fragebaum B veranschaulichen, dessen Wurzel mit der ersten Frage von A markiert ist. Jeder mit einer Frage markierte Knoten hat zwei Kinder, die die Antworten ja und nein auf diese Frage repräsentieren. Stellt A nach Erhalt der Antwort eine weitere Frage, so markieren wir den entsprechenden Antwortknoten mit dieser Frage. Andernfalls gibt A eine Permutation π der Eingabefolge aus und der zugehörige Antwortknoten ist ein Blatt, das wir mit π markieren. Nun ist leicht zu sehen,

dass die Tiefe von B mit der Anzahl $V(n)$ der von A benötigten Fragen im schlechtesten Fall übereinstimmt. Da jede Eingabefolge zu einem anderen Blatt führt, hat B mindestens $n!$ Blätter. Folglich können wir in B einen Pfad der Länge $\lceil \log_2(n!) \rceil$ finden, indem wir jeweils in den Unterbaum mit der größeren Blätterzahl verzweigen.

Da also jedes vergleichende Sortierverfahren mindestens $\Omega(n \log n)$ Fragen benötigt, ist `Mergesort` asymptotisch optimal.

Korollar 12. *`MergeSort` ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $O(n \log n)$.*

2.3.5 QuickSort

Ein weiteres Sortierverfahren, das den "Divide and Conquer"-Ansatz benutzt, ist `QuickSort`. Im Unterschied zu `MergeSort` wird hier das Feld *vor* den rekursiven Aufrufen umsortiert.

Algorithmus `QuickSort`(A, l, r)

```
1  if l < r then m := Partition(A, l, r)
2     QuickSort(A, l, m − 1)
3     QuickSort(A, m + 1, r)
```

Die Prozedur `QuickSort`(A, l, r) sortiert ein Feld $A[l \ldots r]$ wie folgt:

- Zuerst wird die Funktion `Partition`(A, l, r) aufgerufen.
- Diese wählt ein *Pivotelement*, welches sich nach dem Aufruf in $A[m]$ befindet, und sortiert das Feld so um, dass gilt:

$$A[i] \leq A[m] \leq A[j] \text{ für alle } i, j \text{ mit } l \leq i < m < j \leq r. \qquad (*)$$

- Danach werden die beiden Teilfolgen $A[l \ldots m-1]$ und $A[m+1 \ldots r]$ durch jeweils einen rekursiven Aufruf sortiert.

Die Funktion `Partition`(A, l, r) *pivotisiert* das Feld $A[l \ldots r]$, indem

- sie $x = A[r]$ als Pivotelement wählt,
- die übrigen Elemente mit x vergleicht und dabei umsortiert und
- den neuen Index $i + 1$ von x zurückgibt.

Prozedur `Partition`(A, l, r)

```
1  i := l − 1
2  for j := l to r − 1 do
3     if A[j] ≤ A[r] then
4        i := i + 1
5        if i < j then
6           vertausche A[i] und A[j]
7  if i + 1 < r then
8     vertausche A[i + 1] und A[r]
9  return(i+1)
```

Unter der Voraussetzung, dass die Funktion `Partition` korrekt arbeitet, d.h. nach ihrem Aufruf gilt $(*)$, folgt die Korrektheit von `QuickSort` durch einen einfachen Induktionsbeweis über die Länge $n = r - l + 1$ des zu sortierenden Arrays.

Die Korrektheit von `Partition` wiederum folgt leicht aus folgender Invariante für die FOR-Schleife:

$$A[k] \leq A[r] \text{ für } k = l, \ldots, i \text{ und } A[k] > A[r] \text{ für } k = i+1, \ldots, j.$$

Da nämlich nach Ende der FOR-Schleife $j = r - 1$ ist, garantiert die Vertauschung von $A[i+1]$ und $A[r]$ die Korrektheit von `Partition`.

Wir müssen also nur noch die Gültigkeit der Schleifeninvariante nachweisen. Um eindeutig definierte Werte von j vor und nach jeder Iteration der FOR-Schleife zu haben, ersetzen wir sie durch eine semantisch äquivalente WHILE-Schleife:

Prozedur `Partition`(A, l, r)

```
1  i := l − 1
2  j := l − 1
```

```
3  while j < r − 1 do
4    j := j + 1
5    if A[j] ≤ A[r] then
6      i := i + 1
7      if i < j then
8        vertausche A[i] und A[j]
9  if i + 1 < r then
10   vertausche A[i + 1] und A[r]
11 return(i+1)
```

Nun lässt sich die Invariante leicht induktiv beweisen.

Induktionsanfang: Vor Beginn der WHILE-Schleife gilt die Invariante, da i und j den Wert $l-1$ haben.

Induktionsschritt: Zunächst wird j hochgezählt und dann $A[j]$ mit $A[r]$ verglichen.

Im Fall $A[j] \leq A[r]$ wird auch i hochgezählt (d.h. nach Zeile 6 gilt $A[i] > A[r]$). Daher gilt *nach* der Vertauschung in Zeile 8: $A[i] \leq A[r]$ und $A[j] > A[r]$, weshalb die Gültigkeit der Invariante erhalten bleibt.

Im Fall $A[j] > A[r]$ behält die Invariante ebenfalls ihre Gültigkeit, da nur j hochgezählt wird und i unverändert bleibt.

Als nächstes schätzen wir die Laufzeit von `QuickSort` im schlechtesten Fall ab. Dieser Fall tritt ein, wenn sich das Pivotelement nach jedem Aufruf von `Partition` am Rand von A (d.h. $m = l$ oder $m = r$) befindet. Dies führt nämlich dazu, dass `Partition` der Reihe nach mit Feldern der Länge $n, n-1, n-2, \ldots, 1$ aufgerufen wird. Da `Partition` für die Umsortierung eines Feldes der Länge n genau $n-1$ Vergleiche benötigt, führt `QuickSort` insgesamt die maximal mögliche Anzahl

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

von Vergleichen aus. Dieser ungünstige Fall tritt insbesondere dann ein, wenn das Eingabefeld A bereits (auf- oder absteigend) sortiert ist.

Im *besten Fall* zerlegt das Pivotelement das Feld dagegen jeweils in zwei gleich große Felder, d.h. $V(n)$ erfüllt die Rekursion

$$V(n) = \begin{cases} 0, & n = 1, \\ V(\lfloor (n-1)/2 \rfloor) + V(\lceil (n-1)/2 \rceil) + n - 1, & n \geq 2. \end{cases}$$

Diese hat die Lösung $V(n) = n \log_2 n - \Theta(n)$ (vgl. die worst-case Abschätzung bei `MergeSort`).

Es gibt auch Pivotauswahlstrategien, die in linearer Zeit z.B. den Median bestimmen. Dies führt auf eine Variante von `QuickSort` mit einer Laufzeit von $\Theta(n \log n)$ bei *allen* Eingaben. Allerdings ist die Bestimmung des Medians für praktische Zwecke meist zu aufwändig.

Bei der Analyse des Durchschnittsfalls gehen wir von einer zufälligen Eingabepermutation $A[1 \ldots n]$ der Folge $1, \ldots, n$ aus. Dann ist die Anzahl $V(n)$ der Vergleichsanfragen von `QuickSort` eine Zufallsvariable. Wir können $V(n)$ als Summe $\sum_{1 \leq i < j \leq n} X_{ij}$ folgender Indikatorvariablen darstellen:

$$X_{ij} = \begin{cases} 1, & \text{falls die Werte } i \text{ und } j \text{ verglichen werden,} \\ 0, & \text{sonst.} \end{cases}$$

Ob die Werte i und j verglichen werden, entscheidet sich beim ersten Aufruf von `Partition`(A, l, r), bei dem das Pivotelement $x = A[r]$ im Intervall

$$I_{ij} = \{i, \ldots, j\}$$

liegt. Bis zu diesem Aufruf werden die Werte im Intervall I_{ij} nur mit Pivotelementen außerhalb von I_{ij} verglichen und bleiben daher im gleichen Teilfeld $A[l \ldots r]$ beisammen. Ist das erste Pivotelement x in I_{ij} nun nicht gleich i oder j, dann werden i und j zwar mit x verglichen. Das liegt daran dass im Fall $i < x < j$ die Werte i und

j bei diesem Aufruf in zwei verschiedene Teilfelder getrennt werden und daher nicht mehr miteinander verglichen werden.

Die Werte i und j werden also genau dann verglichen, wenn das erste Pivotelement x im Intervall I_{ij} den Wert i oder j hat. Da die Eingabe eine Zufallsfolge ohne Mehrfachvorkommen ist, nimmt x jeden Wert in I_{ij} mit Wahrscheinlichkeit $1/(j-i+1)$ an. Daher findet mit Wahrscheinlichkeit $p_{ij} = 2/(j-i+1)$ ein Vergleich zwischen den Werten i und j statt.

Der Erwartungswert von $V(n) = \sum_{1 \le i < j \le n} X_{ij}$ berechnet sich nun zu

$$\begin{aligned} E[V(n)] &= \sum_{1 \le i < j \le n} \underbrace{E[X_{ij}]}_{p_{ij}} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\le \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n). \end{aligned}$$

Damit haben wir folgenden Satz bewiesen. Dass für vergleichende Sortierverfahren eine Laufzeit von $O(n \log n)$ auch im Durchschnitt asymptotisch optimal ist, wird in den Übungen gezeigt.

Satz 13. *`QuickSort` ist ein vergleichendes Sortierverfahren mit einer im Durchschnitt asymptotisch optimalen Laufzeit von $O(n \log n)$.*

Unabhängig davon nach welcher (deterministischen) Strategie das Pivotelement gewählt wird, wird es immer Eingabefolgen geben, für die `QuickSort` $\binom{n}{2}$ Vergleiche benötigt. Eine Möglichkeit, die Effizienz von `QuickSort` im Durchschnittsfall auf den schlechtesten Fall zu übertragen, besteht darin, eine *randomisierte* Auswahlstrategie für das Pivotelement anzuwenden.

Die Prozedur `RandomQuickSort`(A, l, r) arbeitet ähnlich wie `QuickSort`. Der einzige Unterschied besteht darin, dass als Pivotelement ein zufälliges Element aus dem Feld $A[l \ldots r]$ gewählt wird.

Algorithmus `RandomQuickSort`(A, l, r)

```
1  if l < r then
2    m := RandomPartition(A, l, r)
3    RandomQuickSort(A, l, m − 1)
4    RandomQuickSort(A, m + 1, r)
```

Prozedur `RandomPartition`(A, l, r)

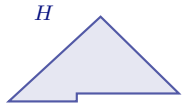
```
1  guess randomly j ∈ {l, ..., r}
2  if j < r then
3    vertausche A[j] und A[r]
4  return(Partition(A, l, r))
```

Es ist nicht schwer zu zeigen, dass sich `RandomQuickSort` bei *jeder* Eingabefolge $A[l, \ldots, r]$ gleich verhält wie `QuickSort` bei einer zufälligen Permutation dieser Eingabefolge (siehe Übungen). Daher ist die erwartete Laufzeit von `RandomQuickSort` auch im *schlechtesten Fall* durch $O(n \log n)$ beschränkt, falls die Zahlenwerte paarweise verschieden sind.

Satz 14. *`RandomQuickSort` ist ein randomisiertes vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $O(n \log n)$.*

2.3.6 HeapSort

`HeapSort` benutzt als Datenstruktur einen so genannten *Heap*, um ein Feld zu sortieren.



Definition 15. *Ein* Heap H *mit n Knoten ist ein geordneter Binärbaum nebenstehender Form. Das heißt,*

- *H hat in Tiefe $i = 0, 1, \ldots, \lfloor \log_2 n \rfloor - 1$ jeweils die maximale Anzahl von 2^i Knoten und*

- *in Tiefe* $\lfloor \log_2 n \rfloor$ *sind alle Knoten linksbündig angeordnet.*

Zudem ist jeder Knoten v mit einer Zahl $H[v]$ beschriftet, deren Wert mindestens so groß ist wie die Werte der Kinder von v (sofern vorhanden).

Ein Heap H mit n Knoten lässt sich in einem Feld $H[1,\ldots,n]$ speichern. Dabei gilt:

- Das linke Kind von Knoten i hat den Index $\texttt{left}(i) = 2i$.
- Das rechte Kind von Knoten i hat den Index $\texttt{right}(i) = 2i+1$.
- Der Elternknoten von Knoten i hat den Index $\texttt{parent}(i) = \lfloor i/2 \rfloor$.

Die Heap-Eigenschaft lässt sich nun wie folgt formulieren. Für alle Knoten $i \in \{1,\ldots,n\}$ gilt

$$(2i \le n \Rightarrow H[i] \ge H[2i]) \wedge (2i+1 \le n \Rightarrow H[i] \ge H[2i+1]).$$

Da die Knoten im Intervall $\{\lfloor n/2 \rfloor + 1,\ldots,n\}$ keine Kinder haben, ist für sie die Heap-Eigenschaft automatisch erfüllt.

Ist $H[1,\ldots,n]$ ein Heap, dann repräsentiert auch jedes Anfangsstück $H[1,\ldots,r]$, $1 \le r \le n$, einen Heap H_r mit r Knoten. Zudem ist für $1 \le i \le r \le n$ der Teilbaum von H_r mit Wurzel i ein Heap, den wir mit $H_{i,r}$ bezeichnen.

Aufgrund der Heap-Eigenschaft muss die Wurzel $H[1]$ eines Heaps den größten Wert haben. Daher können wir eine in einem Heap $H[1,\ldots,n]$ gespeicherte Zahlenfolge sortieren, indem wir sukzessive

- die Wurzel $H[1]$ mit dem letzten Feldelement des Heaps vertauschen,
- den rechten Rand des Heaps um ein Feld nach links verschieben (also die vormalige Wurzel des Heaps herausnehmen) und
- die durch die Ersetzung der Wurzel verletzte Heap-Eigenschaft wieder herstellen.

Natürlich müssen wir zu Beginn die Heap-Eigenschaft auf dem gesamten Feld erst einmal herstellen.

Sei $H[1,\ldots,n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap- Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt. Dann stellt die Prozedur $\texttt{Heapify}(H,i,r)$ die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ her.

Prozedur $\texttt{Heapify}(H,i,r)$

```
1  if (2i ≤ r) ∧ (H[2i] > H[i]) then
2     x := 2i
3  else
4     x := i
5  if (2i + 1 ≤ r) ∧ (H[2i + 1] > H[x]) then
6     x := 2i + 1
7  if x > i then
8     vertausche H[x] und H[i]}
9     Heapify(H, x, r)
```

Unter Verwendung der Prozedur `Heapify` ist es nun leicht, ein Feld zu sortieren.

Algorithmus $\texttt{HeapSort}(H,1,n)$

```
1  for i := ⌊n/2⌋ downto 1 do
2     Heapify(H, i, n)
3  for r := n downto 2 do
4     vertausche H[1] und H[r]
5     Heapify(H, 1, r − 1)
```

Wir setzen zunächst voraus, dass die Prozedur `Heapify` korrekt arbeitet. D.h. $\texttt{Heapify}(H,i,r)$ stellt die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ her, falls $H_{i,r}$ die Heap-Eigenschaft höchstens in seiner Wurzel i nicht erfüllt. Unter dieser Voraussetzung folgt die Korrektheit von `HeapSort` durch den Nachweis folgender Schleifeninvarianten, der sich sehr leicht erbringen lässt.

Invariante für die erste FOR-Schleife:

Für $j = i, \ldots, n$ erfüllt der Teilbaum $H_{j,n}$ die Heap-Eigenschaft.

Invariante für die zweite FOR-Schleife:

$H[r], \ldots, H[n]$ enthalten die $n-r+1$ größten Feldelemente in sortierter Reihenfolge und der Teilbaum $H_{1,r-1}$ erfüllt die Heap-Eigenschaft.

Als nächstes zeigen wir die Korrektheit von `Heapify`. Sei also $H[1, \ldots, n]$ ein Feld, so dass der Teilbaum $H_{i,r}$ die Heap-Eigenschaft in allen Knoten bis auf seine Wurzel i erfüllt. Dann müssen wir zeigen, dass `Heapify`(H, i, r) die Heap-Eigenschaft im gesamten Teilbaum $H_{i,r}$ herstellt.

`Heapify`(H, i, r) bestimmt den Knoten $x \in \{i, 2i, 2i+1\}$ mit maximalem Wert $H(x)$. Im Fall $x = i$ erfüllt der Knoten i bereits die Heap-Eigenschaft. Ist x dagegen eines der Kinder von i, so vertauscht `Heapify` die Werte von i und x. Danach ist die Heap-Eigenschaft höchstens noch im Knoten x verletzt. Daher folgt die Korrektheit von `Heapify` durch einen einfachen Induktionsbeweis über die Rekursionstiefe.

Es ist leicht zu sehen, dass `Heapify`(H, i, r) maximal $2h(i)$ Vergleiche benötigt, wobei $h(i)$ die Höhe des Knotens i in $H_{1,r}$ ist. Daher ist die Laufzeit von `Heapify`(H, i, r) durch $O(h(i)) = O(\log r)$ beschränkt.

Für den Aufbau eines Heaps H der Tiefe $t = \lfloor \log_2 n \rfloor$ wird `Heapify` in der ersten FOR-Schleife für höchstens

- $2^0 = 1$ Knoten der Höhe $h = t$,
- $2^1 = 2$ Knoten der Höhe $h = t - 1$,

  $\vdots$

- 2^{t-1} Knoten der Höhe $h = t - (t-1) = 1$

aufgerufen. Daher benötigt der Aufbau des Heaps maximal

$$V_1(n) \leq 2 \sum_{h=1}^{\lfloor \log_2 n \rfloor} h 2^{\lfloor \log_2 n \rfloor - h} \leq 2 \sum_{h=1}^{\lfloor \log_2 n \rfloor} h \left\lfloor \frac{n}{2^h} \right\rfloor < 2n \sum_{h=1}^{\infty} \frac{h}{2^h} = 4n$$

Vergleiche. Für den Abbau des Heaps in der zweiten FOR-Schleife wird `Heapify` genau $(n-1)$-mal aufgerufen. Daher benötigt der Abbau des Heaps maximal

$$V_2(n) \leq 2(n-1)\lfloor \log_2 n \rfloor \leq 2n \log_2 n$$

Vergleiche.

Satz 16. *`HeapSort` ist ein vergleichendes Sortierverfahren mit einer im schlechtesten Fall asymptotisch optimalen Laufzeit von $O(n \log n)$.*

Die Floyd-Strategie

Die *Floyd-Strategie* benötigt beim Abbau des Heaps im Durchschnitt nur halb so viele Vergleiche wie die bisher betrachtete *Williams-Strategie*. Die Idee besteht darin, dass `Heapify`$(H, 1, r)$ beginnend mit der Wurzel $i_0 = 1$ sukzessive die Werte der beiden Kinder des aktuellen Knotens i_j vergleicht und jeweils zu dem Kind i_{j+1} mit dem größeren Wert absteigt, bis nach $t \leq \lfloor \log_2 n \rfloor$ Schritten ein Blatt i_t erreicht wird.

Nun geht `Heapify` auf diesem Pfad bis zum ersten Knoten i_j mit $H[i_j] \geq H[1]$ zurück und führt auf den Werten der Knoten $i_j, i_{j-1}, \ldots, i_0$ den für die Herstellung der Heap-Eigenschaft erforderlichen Ringtausch aus. Offensichtlich benötigt diese Strategie

$$t + (t - j + 1) = 2t - j + 1$$

Vergleiche (im Unterschied zu höchstens $2j$ Vergleichen bei der Williams-Strategie). Da sich der Knoten i_j, an den der Wert $H[1]$ des Wurzelknotens zyklisch verschoben wird, im Mittel sehr weit unten im Baum befindet, spart man auf diese Art asymptotisch die Hälfte der Vergleiche.

2.3.7 CountingSort

Die Prozedur `CountingSort` sortiert eine Zahlenfolge, indem sie zunächst die Anzahl der Vorkommen jedes Wertes in der Folge und daraus die Rangzahlen der Zahlenwerte bestimmt. Dies funktioniert nur unter der Einschränkung, dass die Zahlenwerte natürliche Zahlen sind und eine Obergrenze k für ihre Größe bekannt ist.

Algorithmus `CountingSort`$(A, 1, n, k)$

```
1  for i := 0 to k do C[i] := 0
2  for j := 1 to n do C[A[j]] := C[A[j]] + 1
3  for i := 1 to k do C[i] := C[i] + C[i − 1]
4  for j := 1 to n do
5     B[C[A[j]]] := A[j]
6     C[A[j]] := C[A[j]] − 1
7  for j := 1 to n do A[j] := B[j]
```

Satz 17. *`CountingSort` sortiert n natürliche Zahlen der Größe höchstens k in Zeit $\Theta(n + k)$ und Platz $\Theta(n + k)$.*

Korollar 18. *`CountingSort` sortiert n natürliche Zahlen der Größe $O(n)$ in linearer Zeit und linearem Platz.*

2.3.8 RadixSort

RadixSort sortiert d-stellige Zahlen $a = a_d \cdots a_1$ eine Stelle nach der anderen, wobei mit der niederwertigsten Stelle begonnen wird.

Algorithmus `RadixSort`$(A, 1, n)$

```
1  for i := 1 to d do
2     sortiere A[1, ..., n] nach der i-ten Stelle
```

Hierzu sollten die Folgenglieder möglichst als Festkomma-Zahlen vorliegen. Zudem muss in Zeile 2 „stabil“ sortiert werden.

Definition 19. *Ein Sortierverfahren heißt* stabil*, wenn es die relative Reihenfolge von Elementen mit demselben Wert nicht verändert.*

Es empfiehlt sich, eine stabile Variante von `CountingSort` als Unterroutine zu verwenden. Damit `CountingSort` stabil sortiert, brauchen wir lediglich die for-Schleife in Zeile 4 in der umgekehrten Reihenfolge zu durchlaufen:

Algorithmus `CountingSort`$(A, 1, n, k)$

```
1  for i := 0 to k do C[i] := 0
2  for j := 1 to n do C[A[j]] := C[A[j]] + 1
3  for i := 1 to k do C[i] := C[i] + C[i − 1]
4  for j := n downto 1 do
5     B[C[A[j]]] := A[j]
6     C[A[j]] := C[A[j]] − 1
7  for j := 1 to n do A[j] := B[j]
```

Satz 20. *`RadixSort` sortiert n d-stellige Festkomma-Zahlen zur Basis b in Zeit $\Theta(d(n + b))$.*

Wenn wir r benachbarte Ziffern zu einer „Ziffer“ $z \in \{0, \ldots, b^r - 1\}$ zusammenfassen, erhalten wir folgende Variante von `RadixSort`.

Korollar 21. *Für jede Zahl $1 \leq r \leq d$ sortiert `RadixSort` n d-stellige Festkomma- Zahlen zur Basis b in Zeit $\Theta(d/r(n + b^r))$.*

Wählen wir beispielsweise $r = \lceil \log_2 n \rceil$, so erhalten wir für $d = O(\log n)$-stellige Binärzahlen eine Komplexität von

$$\Theta\left(d/r(n + 2^r)\right) = \Theta(n + 2^r) = \Theta(n).$$

2.3.9 BucketSort

Die Prozedur `BucketSort` sortiert n Zahlen $a_1, \ldots, a_n$ aus einem Intervall $[a, b)$ wie folgt. Der Einfachheit halber nehmen wir $a = 0$ und $b = 1$ an.

1. Erstelle für $j = 0, \ldots, n-1$ eine Liste L_j für das halb offene Intervall $I_j = [j/n, (j+1)/n)$.
2. Bestimme zu jedem Element a_i das Intervall I_j, zu dem es gehört, und füge es in die entsprechende Liste L_j ein.
3. Sortiere jede Liste L_j.
4. Füge die sortierten Listen L_j wieder zu einer Liste zusammen.

Um Listen L_j mit einer konstanten erwarteten Länge zu erhalten, sollten die zu sortierenden Zahlenwerte im Intervall $[a, b)$ möglichst gleichverteilt sein. In diesem Fall ist die erwartete Laufzeit von `BucketSort` nämlich $\Theta(n)$. Wie wir gleich zeigen werden, gilt dies sogar, wenn als Unterroutine ein Sortierverfahren der Komplexität $O(n^2)$ verwendet wird. Im schlechtesten Fall kommen dagegen alle Schlüssel in die gleiche Liste. Dann hat `BucketSort` dieselbe asymptotische Laufzeit wie das als Unterroutine verwendete Sortierverfahren.

Wir schätzen nun die erwartete Laufzeit von `BucketSort` ab. Sei X_i die Zufallsvariable, die die Länge der Liste L_i beschreibt. Dann ist X_i binomialverteilt mit Parametern n und $p = 1/n$. Also hat X_i den Erwartungswert

$$E[X_i] = np = 1$$

und die Varianz

$$V[X_i] = np(1-p) = 1 - 1/n < 1.$$

Wegen $V[X_i] = E[X_i^2] - E[X_i]^2$ ist $E[X_i^2] = V[X_i] + E[X_i]^2 < 2$. Daher folgt für die erwartete Laufzeit von `BucketSort`:

$$E\left[\sum_{i=0}^{n-1} O(X_i^2)\right] = O\left(\sum_{i=0}^{n-1} E[X_i^2]\right) = O(n).$$

2.3.10 Vergleich der Sortierverfahren

Folgende Tabelle zeigt die Komplexitäten der betrachteten vergleichenden Sortierverfahren.

	`Insertion-Sort`	`MergeSort`	`Quick-Sort`	`Heap-Sort`
worst-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
average-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Speicherplatz	$\Theta(1)$	$\Theta(n)$ bzw. $\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
stabil	ja	ja	nein	nein

Wir fassen auch die wichtigsten Eigenschaften der betrachteten Linearzeit-Sortierverfahren zusammen.

- `CountingSort`: lineare Zeit im schlechtesten Fall, falls die Werte natürliche Zahlen sind und $O(n)$ nicht übersteigen.
- `RadixSort`: bitweises Sortieren in linearer Zeit, falls die zu sortierenden Zahlen nur $O(\log n)$ Bit haben.
- `BucketSort`: Sortieren in erwarteter linearer Zeit, falls die n Zahlen gleichmäßig über einem Intervall $[a, b)$ verteilt sind.

2.4 Datenstrukturen für dynamische Mengen

Eine Datenstruktur für dynamische Mengen S sollte im Prinzip *beliebig viele* Elemente aufnehmen können. Die Elemente $x \in S$ werden dabei anhand eines *Schlüssels* $k = \texttt{key}(x)$ identifiziert. Auf die Elemente $x \in S$ wird meist nicht direkt, sondern mittels *Zeiger* (engl. *pointer*) zugegriffen.

Typische Operationen, die auf einer dynamische Mengen S auszuführen sind:

Insert(S, x)**:** Füge x in S ein.

Remove(S, x)**:** Entferne x aus S.

Search(S, k)**:** Gib für einen Schlüssel k einen Zeiger auf $x \in S$ mit $\texttt{key}(x) = k$, falls ein solches Element existiert, und sonst `nil` zurück.

Min(S)**:** Gib einen Zeiger auf das Element in S mit dem kleinsten Schlüssel zurück.

Max(S)**:** Gib einen Zeiger auf das Element in S mit dem größten Schlüssel zurück.

Prec(S, x)**:** Gib einen Zeiger auf das nach x nächstkleinere Element in S oder `nil` zurück, falls x das Minimum ist.

Succ(S, x)**:** Gib einen Zeiger auf das nach x nächstgrößere Element in S oder `nil` zurück, falls x das Maximum ist.

2.4.1 Verkettete Listen

Die Elemente einer verketteten Liste sind in linearer Reihenfolge angeordnet. Das erste Element der Liste L ist $\texttt{head}(L)$. Jedes Element x „kennt“ seinen Nachfolger $\texttt{next}(x)$. Wenn jedes Element x auch seinen Vorgänger $\texttt{prev}(x)$ kennt, dann spricht man von einer *doppelt verketteten Liste*.

Die Prozedur `DL-Insert`(L, x) fügt ein Element x in eine doppelt verkettete Liste L ein.

Prozedur `DL-Insert`(L, x)

```
1 next(x) := head(L)
2 if head(L) ≠ nil then
3   prev(head(L)) := x
4 head(L) := x
5 prev(x) := nil
```


Die Prozedur `DL-Remove`(L, x) entfernt wieder ein Element x aus einer doppelt verketteten Liste L.

Prozedur `DL-Remove`(L, x)

```
1 if x ≠ head(L) then
2   next(prev(x)) := next(x)
3 else
4   head(L) := next(x)
5 if next(x) ≠ nil then
6   prev(next(x)) := prev(x)
```

Die Prozedur `DL-Search`(L, k) sucht ein Element x mit dem Schlüssel k in der Liste L.

Prozedur `DL-Search`(L, k)

```
1 x := head(L)
2 while x ≠ nil and key(x) ≠ k do
3   x := next(x)
4 return(x)
```

Es ist leicht zu sehen, dass `DL-Insert` und `DL-Remove` konstante Zeit $\Theta(1)$ benötigen, während `DL-Search` eine lineare (in der Länge der Liste) Laufzeit hat.

Bemerkung 22.

- *Wird* `DL-Remove` *nur der Schlüssel übergeben, dann wäre die Laufzeit linear, da wir erst mit* `DL-Search` *das entsprechende Element suchen müssen.*
- *Für einfach verkettete Listen ist der Aufwand von* `Remove` *ebenfalls linear, da wir keinen direkten Zugriff auf den Vorgänger haben.*
- *Die Operationen* `Max`*,* `Min`*,* `Prec` *und* `Succ` *lassen sich ebenfalls mit linearer Laufzeit berechnen (siehe Übungen).*

- `MergeSort` *lässt sich für Listen sogar als "in place"-Verfahren implementieren (siehe Übungen). Daher lassen sich Listen in Zeit* $O(n \log n)$ *sortieren.*

2.4.2 Binäre Suchbäume

Ein Binärbaum B kann wie folgt durch eine Zeigerstruktur repräsentiert werden. Jeder Knoten x in B hat 3 Zeiger:

- $\texttt{left}(x)$ zeigt auf das linke Kind,
- $\texttt{right}(x)$ zeigt auf das rechte Kind und
- $\texttt{parent}(x)$ zeigt auf den Elternknoten.

Für die Wurzel $w = \texttt{root}(B)$ ist $\texttt{parent}(w) = \texttt{nil}$ und falls einem Knoten x eines seiner Kinder fehlt, so ist der entsprechende Zeiger ebenfalls `nil`. Auf diese Art lassen sich beispielsweise Heaps für unbeschränkt viele Datensätze implementieren.

Definition 23. *Ein binärer Baum B ist ein* binärer Suchbaum, *falls für jeden Knoten x in B folgende Eigenschaften erfüllt sind:*

- *Für jeden Knoten y im linken Teilbaum von x gilt* $\texttt{key}(y) \leq \texttt{key}(x)$ *und*
- *für jeden Knoten y im rechten Teilbaum von x gilt* $\texttt{key}(y) \geq \texttt{key}(x)$.

Folgende Prozedur $\texttt{ST-Search}(B, k)$ sucht ein Element x mit dem Schlüssel k im binären Suchbaum (engl. *search tree*) B.

Prozedur $\texttt{ST-Search}(B, k)$

```
1  x := root(B)
2  while x ≠ nil ∧ key(x) ≠ k do
3     if k ≤ key(x) then
4        x := left(x)
5     else
6        x := right(x)
7  return(x)
```

Die Prozedur $\texttt{ST-Insert}(B, z)$ fügt ein neues Element z in B ein, indem sie den `nil`-Zeiger „sucht", der eigentlich auf den Knoten z zeigen müsste.

Prozedur $\texttt{ST-Insert}(B, z)$

```
1  if root(B) = nil then
2     root(B) := z
3     parent(z) := nil
4  else
5     x := root(B)
6     repeat
7        y := x
8        if key(z) ≤ key(x) then
9           x := left(x)
10       else
11          x := right(x)
12    until x = nil
13    if key(z) ≤ key(y) then
14       left(y) := z
15    else
16       right(y) := z
17    parent(z) := y
```

Satz 24. *Die Prozeduren* `ST-Search` *und* `ST-Insert` *laufen auf einem binären Suchbaum der Höhe h in Zeit $O(h)$.*

Bemerkung 25. *Auch die Operationen* `Min`, `Max`, `Succ`, `Prec` *und* `Remove` *lassen sich auf einem binären Suchbaum der Höhe h in Zeit $O(h)$ implementieren (siehe Übungen).*

Die Laufzeiten der Operationen für binäre Suchbäume hängen von der Tiefe der Knoten im Suchbaum ab. Suchbäume können zu Listen

entarten. Dieser Fall tritt z.B. ein, falls die Datensätze in sortierter Reihenfolge eingefügt werden. Daher haben die Operationen im schlechtesten Fall eine lineare Laufzeit.

Für die Analyse des Durchschnittsfalls gehen wir davon aus, dass die Einfügesequenz eine zufällige Permutation von n verschiedenen Zahlen ist. Dann lässt sich zeigen, dass der resultierende Suchbaum eine erwartete Tiefe von $O(\log n)$ hat (siehe Übungen). Somit ist die erwartete Laufzeit der Operationen nur $O(\log n)$.

2.4.3 Balancierte Suchbäume

Um die Tiefe des Suchbaums klein zu halten, kann er während der Einfüge- und Löschoperationen auch aktiv ausbalanciert werden. Hierfür gibt es eine ganze Reihe von Techniken. Die drei bekanntesten sind Rot-Schwarz-Bäume, Splay-Bäume und die AVL-Bäume, mit denen wir uns im Folgenden etwas näher befassen möchten.

Definition 26. *Ein* AVL-Baum T *ist ein binärer Suchbaum, der* höhenbalanciert *ist, d.h. für jeden Knoten x von T unterscheiden sich die Höhen des linken und rechten Teilbaumes von x höchstens um eins (die Höhe eines nicht existierenden Teilbaumes setzen wir mit -1 an).*

Lemma 27. *Die Höhe eines AVL-Baumes mit n Knoten ist $O(\log n)$.*

Beweis. Sei $M(h)$ die minimale Blattzahl eines AVL-Baumes der Höhe h. Dann gilt

$$M(h) = \begin{cases} 1, & h = 0 \text{ oder } 1, \\ M(h-1) + M(h-2), & h \geq 2. \end{cases}$$

$M(h)$ ist also die $(h+1)$-te *Fibonacci-Zahl* F_{h+1}. Durch Induktion über h lässt sich leicht zeigen, dass $F_{h+1} \geq \phi^{h-1}$ ist, wobei $\phi = (1+\sqrt{5})/2$ der *goldene Schnitt* ist. Daher folgt für einen AVL-Baum der Höhe h mit n Knoten und b Blättern

$$n \geq b \geq M(h) = F_{h+1} \geq \phi^{h-1}.$$

Somit gilt

$$h \leq 1 + \log_\phi(n) = O(\log_2 n).$$

Der konstante Faktor in $O(\log_2 n)$ ist hierbei $\frac{1}{\log_2(\phi)} \approx 1{,}44$. □

Für die Aufrechterhaltung der AVL-Eigenschaft eines AVL-Baums T benötigen wir folgende Information über jeden Knoten x. Seien h_l und h_r die Höhen des linken und des rechten Teilbaums von x. Dann heißt die Höhendifferenz

$$\texttt{bal}(x) = h_l - h_r$$

die *Balance* von x in T. T ist also genau dann ein AVL-Baum, wenn jeder Knoten x in T eine Höhendifferenz zwischen -1 und 1 hat:

$$\forall x : \texttt{bal}(x) \in \{-1, 0, 1\}.$$

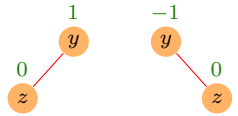
Im Folgenden bezeichne $T(x)$ den Teilbaum von T mit der Wurzel x.

Wir fügen einen neuen Knoten z in einen AVL-Baum T ähnlich wie die Prozedur `ST-Insert` für binäre Suchbäume ein. D.h. wir „suchen“ den Schlüssel $k = \texttt{key}(z)$ in T bis wir einen Knoten y mit $k \leq \texttt{key}(y)$ und $\texttt{left}(y) = \texttt{nil}$ bzw. $k > \texttt{key}(y)$ und $\texttt{right}(y) = \texttt{nil}$ erreichen und fügen z an dieser Stelle als Kind von y ein. Da z ein Blatt ist, erhält z den Wert $\texttt{bal}(z) = 0$. Das Einfügen von z kann nur für Knoten auf dem Pfad von z zur Wurzel von T eine Änderung der Höhendifferenz bewirken. Daher genügt es, diesen Suchpfad zurückzugehen und dabei für jeden besuchten Knoten die AVL-Eigenschaft zu testen und nötigenfalls wiederherzustellen.

Wir untersuchen zuerst, ob y die AVL-Eigenschaft verletzt.

1. Falls der Wert von $\texttt{bal}(y)$ gleich -1 oder 1 ist, hatte $T(y)$ schon vor dem Einfügen von z die Höhe 1. Daher genügt es, $\texttt{bal}(y) = 0$ zu setzen.

2. Falls $\texttt{bal}(y) = 0$ ist, wurde z an ein Blatt gehängt, d.h. die Höhe von $T(y)$ ist um 1 gewachsen. Zunächst setzen wir $\texttt{bal}(y)$ auf den Wert 1 oder -1,



je nachdem ob z linkes oder rechtes Kind von y ist. Dann wird die rekursive Prozedur $\texttt{AVL-Check-Insertion}(y)$ aufgerufen, die überprüft, ob weitere Korrekturen nötig sind.

Prozedur $\texttt{AVL-Insert}(B, z)$

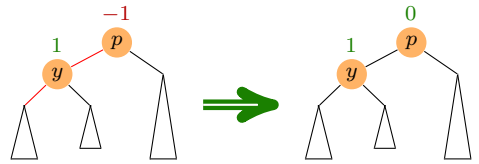
```
1  if root(B) = nil then
2     root(B) := z
3     parent(z) := nil
4     bal(z) := 0
5  else
6     x := root(B)
7     repeat
8        y := x
9        if key(z) ≤ key(x) then
10          x := left(x)
11       else
12          x := right(x)
13    until (x = nil)
14    if key(z) ≤ key(y) then
15       left(y) := z
16    else
17       right(y) := z
18    parent(z) := y
19    bal(z) := 0
20    if bal(y) ∈ {−1, 1} then
21       bal(y) := 0
22    else
23       if z = left(y) then
24          bal(y) := 1
25       else
26          bal(y) := −1
27       AVL-Check-Insertion(y)
```


Als nächstes beschreiben wir die Arbeitsweise der Prozedur $\texttt{AVL-Check- Insertion}(y)$. Dabei setzen wir voraus, dass bei jedem Aufruf von $\texttt{AVL- Check-Insertion}(y)$ folgende Invariante gilt:

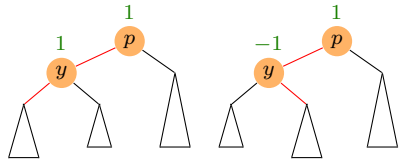
> Der Wert von $\texttt{bal}(y)$ wurde von 0 auf ± 1 aktualisiert, d.h. die Höhe von $T(y)$ ist um 1 gewachsen.

Falls y die Wurzel von T ist, ist nichts weiter zu tun. Andernfalls nehmen wir an, dass y linkes Kind von $p = \texttt{parent}(y)$ ist (der Fall $y = \texttt{right}(p)$ ist analog).

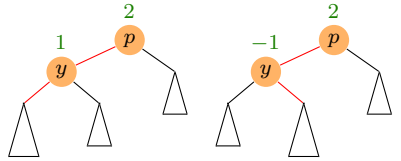
1. Im Fall $\texttt{bal}(p) = -1$ genügt es, $\texttt{bal}(p) = 0$ zu setzen.



2. Im Fall $\texttt{bal}(p) = 0$ setzen wir $\texttt{bal}(p) = 1$ und rufen $\texttt{AVL-Check- Insertion}(p)$ auf.



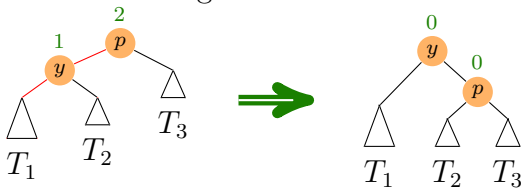
3. Im Fall $\texttt{bal}(p) = 1$ müssen wir T umstrukturieren, da die aktuelle Höhendifferenz von p gleich 2 ist.



   3a. Im Fall $\texttt{bal}(y) = 1$ sei T_1 der linke und T_2 der rechte Teilbaum von y. Weiter sei T_3 der rechte Teilbaum von p und h sei die Höhe von $T(y)$. Dann gilt für die Höhen h_i der Teilbäume T_i:

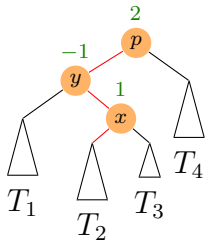
$$h_1 = h - 1 \text{ und } h_2 = h_3 = h - 2.$$

Wir führen nun eine so genannte *Rechts-Rotation* aus,



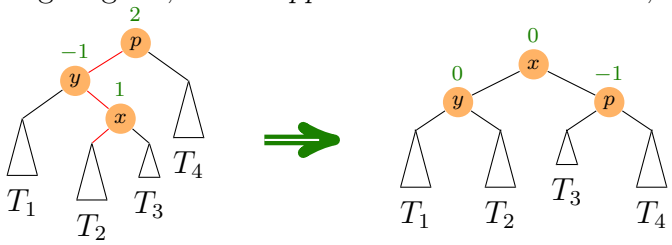
d.h. p wird rechtes Kind von y und erhält T_2 als linken und T_3 als rechten Teilbaum (d.h. $\texttt{bal}(p)$ erhält den Wert 0) und T_1 bleibt linker Teilbaum von y (d.h. $\texttt{bal}(y)$ erhält ebenfalls den Wert 0). Dann hat der rotierte Teilbaum wieder die gleiche Höhe wie vor dem Einfügen von z. Daher ist nichts weiter zu tun.

3b. Im Fall $\texttt{bal}(y) = -1$ sei T_1 der linke Teilbaum von y und T_4 der rechte Teilbaum von p. Weiter seien T_2 und T_3 der linke und rechte Teilbaum von $x = \texttt{right}(y)$. Die Höhe von $T(y)$ bezeichnen wir wieder mit h. Dann gilt



$h_1 = h_4 = h-2$ und $h-3 \leq h_2, h_3 \leq h-2$,
wobei $h_3 = h_2 - \texttt{bal}(x)$ ist und $\texttt{bal}(x) = 0$ nur im Fall $h = 1$ (d.h. alle Teilbäume T_1, T_2, T_3 und T_4 sind leer) möglich ist.

Daher genügt es, eine *Doppel-Rotation* auszuführen,



d.h. y wird linkes und p wird rechtes Kind von x, y erhält T_1 als linken und T_2 als rechten Teilbaum und p erhält T_3 als linken und T_4 als rechten Teilbaum. Die neuen Balance-Werte von p, y und x sind

$$\texttt{bal}(p) = \begin{cases} -1, & \texttt{bal}(x) = 1, \\ 0, & \text{sonst}, \end{cases} \qquad \texttt{bal}(y) = \begin{cases} 1, & \texttt{bal}(x) = -1, \\ 0, & \text{sonst} \end{cases}$$

und $\texttt{bal}(x) = 0$. Der rotierte Teilbaum hat die gleiche Höhe wie der ursprüngliche Teilbaum an dieser Stelle und daher ist nichts weiter zu tun.

Wir fassen die worst-case Komplexitäten der betrachteten Datenstrukturen für dynamische Mengen in folgender Tabelle zusammen.

	Search	Min/Max	Prec/Succ	Insert	Remove
Heap	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Listeq (einfach)	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Liste (doppelt)	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Suchbaum	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL-Baum	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

3 Graphalgorithmen

3.1 Grundlegende Begriffe

Definition 28. *Ein* (ungerichteter) Graph *ist ein Paar* $G = (V, E)$, *wobei*

V - *eine endliche Menge von* Knoten/Ecken *und*

E - *die Menge der* Kanten *ist.*

Hierbei gilt

$$E \subseteq \binom{V}{2} = \Big\{\{u, v\} \subseteq V \mid u \neq v\Big\}.$$

Sei $v \in V$ *ein Knoten.*

a) Die Nachbarschaft *von* v *ist* $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$.

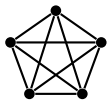
b) Der Grad *von* v *ist* $\deg_G(v) = \|N_G(v)\|$.

c) Der Minimalgrad *von* G *ist* $\delta(G) = \min_{v \in V} \deg_G(v)$ *und der* Maximalgrad *von* G *ist* $\Delta(G) = \max_{v \in V} \deg_G(v)$.

Falls G aus dem Kontext ersichtlich ist, schreiben wir auch einfach $N(v)$, $\deg(v)$, δ usw.

Beispiel 29.

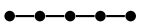
- *Der* vollständige Graph (V, E) *auf* n *Knoten, d.h.* $\|V\| = n$ *und* $E = \binom{V}{2}$, *wird mit* K_n *und der* leere Graph $(V, \emptyset)$ *auf* n *Knoten wird mit* E_n *bezeichnet.*

K_1: K_2: K_3:  K_4:  K_5: 

- *Der* vollständige bipartite Graph (A, B, E) *auf* $a+b$ *Knoten, d.h.* $A \cap B = \emptyset$, $\|A\| = a$, $\|B\| = b$ *und* $E = \{\{u, v\} \mid u \in A, v \in B\}$ *wird mit* $K_{a,b}$ *bezeichnet.*

$K_{1,1}$:  $K_{1,2}$:  $K_{2,2}$:  $K_{2,3}$:  $K_{3,3}$: 

- *Der* Pfad der Länge n *wird mit* P_n *bezeichnet.*

P_1:  P_2:  P_3: P_4: 

- *Der* Kreis der Länge n *wird mit* C_n *bezeichnet.*

C_3:  C_4:  C_5:  C_6: 

Definition 30. *Sei* $G = (V, E)$ *ein Graph.*

a) Eine Knotenmenge $U \subseteq V$ *heißt* stabil, *wenn es keine Kante von* G *mit beiden Endpunkten in* U *gibt, d.h. es gilt* $E \cap \binom{U}{2} = \emptyset$. *Die* Stabilitätszahl *ist*

$$\alpha(G) = \max\{\|U\| \mid U \text{ ist stabile Menge in } G\}.$$

b) Eine Knotenmenge $U \subseteq V$ *heißt* Clique, *wenn jede Kante mit beiden Endpunkten in* U *in* E *ist, d.h. es gilt* $\binom{U}{2} \subseteq E$. *Die* Cliquenzahl *ist*

$$\omega(G) = \max\{\|U\| \mid U \text{ ist Clique in } G\}.$$

c) Eine Abbildung $f\colon V \to \mathbb{N}$ *heißt* Färbung *von* G, *wenn* $f(u) \neq f(v)$ *für alle* $\{u, v\} \in E$ *gilt.* G *heißt* k-färbbar, *falls eine Färbung* $f\colon V \to \{1, \ldots, k\}$ *existiert. Die* chromatische Zahl *ist*

$$\chi(G) = \min\{k \in \mathbb{N} \mid G \text{ ist } k\text{-färbbar}\}.$$

d) Ein Graph heißt bipartit, *wenn* $\chi(G) \leq 2$ *ist.*

e) Ein Graph $G' = (V', E')$ heißt Sub-/Teil-/Untergraph *von G, falls $V' \subseteq V$ und $E' \subseteq E$ ist. Ein Subgraph $G' = (V', E')$ heißt (*durch V'*)* induziert*, falls $E' = E \cap \binom{V'}{2}$ ist. Hierfür schreiben wir auch $H = G[V']$.*

f) Ein Weg *ist eine Folge von (nicht notwendig verschiedenen) Knoten $v_0, \ldots, v_\ell$ mit $\{v_i, v_{i+1}\} \in E$ für $i = 0, \ldots, \ell - 1$. Die* Länge *des Weges ist die Anzahl der Kanten, also ℓ. Im Fall $\ell = 0$ heißt der Weg* trivial*. Ein Weg $v_0, \ldots, v_\ell$ heißt auch v_0-v_ℓ-*Weg*.*

g) Ein Graph $G = (V, E)$ heißt zusammenhängend*, falls es für alle Paare $\{u, v\} \in \binom{V}{2}$ einen u-v-Weg gibt.*

h) Ein Zyklus *ist ein u-v-Weg der Länge $\ell \geq 2$ mit $u = v$.*

i) Ein Weg heißt einfach *oder* Pfad*, falls alle durchlaufenen Knoten verschieden sind.*

j) Ein Kreis *ist ein Zyklus $v_0, v_1 \ldots, v_{\ell-1}, v_0$ der Länge $\ell \geq 3$, für den $v_0, v_1, \ldots, v_{\ell-1}$ paarweise verschieden sind.*

k) Ein Graph $G = (V, E)$ heißt kreisfrei*,* azyklisch *oder* Wald*, falls er keinen Kreis enthält.*

l) Ein Baum *ist ein zusammenhängender Wald.*

*m) Jeder Knoten $u \in V$ vom Grad $\deg(u) \leq 1$ heißt **Blatt** und die übrigen Knoten (vom Grad ≥ 2) heißen **innere Knoten**.*

Es ist leicht zu sehen, dass die Relation

$$Z = \{(u, v) \in V \times V \mid \text{ es gibt in } G \text{ einen } u\text{-}v\text{-Weg}\}$$

eine Äquivalenzrelation ist. Die durch die Äquivalenzklassen von Z induzierten Teilgraphen heißen die *Zusammenhangskomponenten* (engl. *connected components*) von G.

Definition 31. *Ein* gerichteter Graph *oder* Digraph *ist ein Paar $G = (V, E)$, wobei*

V - eine endliche Menge von Knoten/Ecken *und*

E - die Menge der Kanten *ist.*

Hierbei gilt

$$E \subseteq V \times V = \Big\{(u, v) \mid u, v \in V\Big\},$$

wobei E auch Schlingen (u, u) enthalten kann. Sei $v \in V$ ein Knoten.

a) Die Nachfolgermenge *von v ist $N^+(v) = \{u \in V \mid (v, u) \in E\}$.*

b) Die Vorgängermenge *von v ist $N^-(v) = \{u \in V \mid (u, v) \in E\}$.*

c) Die Nachbarmenge *von v ist $N(v) = N^+(v) \cup N^-(v)$.*

d) Der Ausgangsgrad *von v ist $\deg^+(v) = \|N^+(v)\|$ und der* Eingangsgrad *von v ist $\deg^-(v) = \|N^-(v)\|$. Der* Grad *von v ist $\deg(v) = \deg^+(v) + \deg^-(v)$.*

e) Ein gerichteter v_0-v_ℓ-Weg *ist eine Folge von Knoten $v_0, \ldots, v_\ell$ mit $(v_i, v_{i+1}) \in E$ für $i = 0, \ldots, \ell - 1$.*

f) Ein gerichteter Zyklus *ist ein gerichteter u-v-Weg der Länge $\ell \geq 1$ mit $u = v$.*

g) Ein gerichteter Weg heißt einfach *oder* gerichteter Pfad*, falls alle durchlaufenen Knoten verschieden sind.*

h) Ein gerichteter Kreis *ist ein gerichteter Zyklus $v_0, v_1 \ldots, v_{\ell-1}, v_0$ der Länge $\ell \geq 1$, für den $v_0, v_1, \ldots, v_{\ell-1}$ paarweise verschieden sind.*

i) Ein Digraph $G = (V, E)$ heißt kreisfrei *oder* azyklisch*, wenn er keinen gerichteten Kreis hat.*

j) Ein Digraph $G = (V, E)$ heißt schwach zusammenhängend*, wenn es für jedes Paar $\{u, v\} \in \binom{V}{2}$ einen gerichteten u-v-Pfad oder einen gerichteten v-u-Pfad gibt.*

k) $G = (V, E)$ heißt stark zusammenhängend*, wenn es für jedes Paar $\{u, v\} \in \binom{V}{2}$ sowohl einen gerichteten u-v-Pfad als auch einen gerichteten v-u-Pfad gibt.*

3.2 Datenstrukturen für Graphen

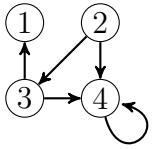
Sei $G = (V, E)$ ein Graph bzw. Digraph und sei $V = \{v_1, \ldots, v_n\}$. Dann ist die $(n \times n)$-Matrix $A = (a_{ij})$ mit den Einträgen

$$a_{ij} = \begin{cases} 1, & \{v_i, v_j\} \in E \\ 0, & \text{sonst} \end{cases} \quad \text{bzw.} \quad a_{ij} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & \text{sonst} \end{cases}$$

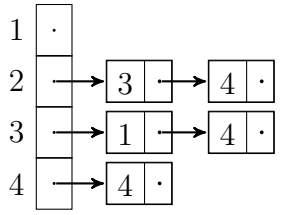
die *Adjazenzmatrix* von G. Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch mit $a_{ii} = 0$ für $i = 1, \ldots, n$.

Bei der *Adjazenzlisten-Darstellung* wird für jeden Knoten v_i eine Liste mit seinen Nachbarn verwaltet. Im gerichteten Fall verwaltet man entweder nur die Liste der Nachfolger oder zusätzlich eine weitere für die Vorgänger. Falls die Anzahl der Knoten gleichbleibt, organisiert man die Adjazenzlisten in einem Feld, d.h. das Feldelement mit Index i verweist auf die Adjazenzliste von Knoten v_i. Falls sich die Anzahl der Knoten dynamisch ändert, so werden die Adjazenzlisten typischerweise ebenfalls in einer doppelt verketteten Liste verwaltet.

Beispiel 32.
Betrachte den gerichteten Graphen $G = (V, E)$ mit $V = \{1, 2, 3, 4\}$ und $E = \{(2, 3), (2, 4), (3, 1), (3, 4), (4, 4)\}$. Dieser hat folgende Adjazenzmatrix- und Adjazenzlisten-Darstellung:



	1	2	3	4
1	0	0	0	0
2	0	0	1	1
3	1	0	0	1
4	0	0	0	1



◁

Folgende Tabelle gibt den Aufwand der wichtigsten elementaren Operationen auf Graphen in Abhängigkeit von der benutzten Datenstruktur an. Hierbei nehmen wir an, dass sich die Knotenmenge V nicht ändert.

	Adjazenzmatrix		Adjazenzlisten	
	einfach	clever	einfach	clever
Speicherbedarf	$O(\|V\|^2)$	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
Initialisieren	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$	$O(1)$
Kante einfügen	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Kante entfernen	$O(1)$	$O(1)$	$O(\|V\|)$	$O(1)$
Test auf Kante	$O(1)$	$O(1)$	$O(\|V\|)$	$O(\|V\|)$

Bemerkung 33.

- *Der Aufwand für die Initialisierung des leeren Graphen in der Adjazenzmatrixdarstellung lässt sich auf $O(1)$ drücken, indem man mithilfe eines zusätzlichen Feldes B die Gültigkeit der Matrixeinträge verwaltet (siehe Übungen).*
- *Die Verbesserung beim Löschen einer Kante in der Adjazenzlistendarstellung erhält man, indem man die Adjazenzlisten doppelt verkettet und im ungerichteten Fall die beiden Vorkommen jeder Kante in den Adjazenzlisten der beiden Endknoten gegenseitig verlinkt (siehe Übungen).*
- *Bei der Adjazenzlistendarstellung können die Knoten auch in einer doppelt verketteten Liste organisiert werden. In diesem Fall können dann auch Knoten in konstanter Zeit hinzugefügt und in Zeit $O(|V|)$ wieder entfernt werden (unter Beibehaltung der übrigen Speicher- und Laufzeitschranken).*

3.3 Keller und Warteschlange

Für das Durchsuchen eines Graphen ist es vorteilhaft, die bereits besuchten (aber noch nicht abgearbeiteten) Knoten in einer Menge B zu speichern. Damit die Suche effizient ist, sollte die Datenstruktur für B folgende Operationen effizient implementieren.

`Init`(B):	Initialisiert B als leere Menge.
`Insert`(B, u):	Fügt u in B ein.
`Empty`(B):	Testet B auf Leerheit.
`Element`(B):	Gibt ein Element aus B zurück.
`Remove`(B):	Gibt ebenfalls `Element`(B) zurück und entfernt es aus B.

Andere Operationen wie z.B. `Remove`(B, u) werden nicht benötigt.

Die gewünschten Operationen lassen sich leicht durch einen *Keller* (auch *Stapel* genannt) (engl. *stack*) oder eine *Warteschlange* (engl. *queue*) implementieren. Falls maximal n Datensätze gespeichert werden müssen, kann ein Feld zur Speicherung der Elemente benutzt werden. Andernfalls können sie auch in einer einfach verketteten Liste gespeichert werden.

Stack S – Last-In-First-Out

`Top`(S):	Gibt das oberste Element von S zurück.
`Push`(S, x):	Fügt x als oberstes Element zum Keller hinzu.
`Pop`(S):	Gibt das oberste Element von S zurück und entfernt es.

Queue Q – Last-In-Last-Out

`Enqueue`(Q, x):	Fügt x am Ende der Schlange hinzu.
`Head`(Q):	Gibt das erste Element von Q zurück.
`Dequeue`(Q):	Gibt das erste Element von Q zurück und entfernt es.

Die Kelleroperationen lassen sich wie folgt auf einem Feld $S[1 \ldots n]$ implementieren. Die Variable `size`(S) enthält die Anzahl der im Keller gespeicherten Elemente.

Prozedur `Init`(S)

```
1  size(S) := 0
```

Prozedur `Push`(S, x)

```
1  if size(S) < n then
2     size(S) := size(S) + 1
3     S[size(S)] := x
4  else
5     return(nil)
```

Prozedur `Empty`(S)

```
1  return(size(S) = 0)
```

Prozedur `Top`(S)

```
1  if size(S) > 0 then
2     return(S[size(S)])
3  else
4     return(nil)
```

Prozedur `Pop`(S)

```
1  if size(S) > 0 then
2     size(S) := size(S) − 1
3     return(S[size(S) + 1])
4  else
5     return(nil)
```

Es folgen die Warteschlangenoperationen für die Speicherung in einem Feld $Q[1 \ldots n]$. Die Elemente werden der Reihe nach am Ende der Schlange Q (zyklisch) eingefügt und am Anfang entnommen. Die Variable `head`(Q) enthält den Index des ersten Elements der Schlange und `tail`(Q) den Index des hinter dem letzten Element von Q befindlichen Eintrags.

Prozedur `Init`(Q)

```
1 head(Q) := 1
2 tail(Q) := 1
3 size(Q) := 0
```

Prozedur `Enqueue`(Q, x)

```
1 if size(Q) = n then
2   return(nil)
3 size(Q) := size(Q) + 1
4 Q[tail(Q)] := x
5 if tail(Q) = n then
6   tail(Q) := 1
7 else
8   tail(Q) := tail(Q) + 1
```

Prozedur `Empty`(Q)

```
1 return(size(Q) = 0)
```

Prozedur `Head`(Q)

```
1 if Empty(Q) then
2   return(nil)
3 else
4   returnQ[head(Q)]
```

Prozedur `Dequeue`(Q)

```
1 if Empty(Q) then
2   return(nil)
3 size(Q) := size(Q) − 1
4 x := Q[head(Q)]
5 if head(Q) = n then
6   head(Q) := 1
7 else
8   head(Q) := head(Q) + 1
9 return(x)
```

Satz 34. *Sämtliche Operationen für einen Keller S und eine Warteschlange Q sind in konstanter Zeit $O(1)$ ausführbar.*

Bemerkung 35. *Mit Hilfe von einfach verketteten Listen sind Keller und Warte- schlangen auch für eine unbeschränkte Anzahl von Datensätzen mit denselben Laufzeitbeschränkungen implementierbar.*

Die für das Durchsuchen von Graphen benötigte Datenstruktur B lässt sich nun mittels Keller bzw. Schlange wie folgt realisieren.

Operation	Keller S	Schlange Q
`Init`(B)	`Init`(S)	`Init`(Q)
`Insert`(B, u)	`Push`(S, u)	`Enqueue`(Q, u)
`Empty`(B)	`Empty`(S)	`Empty`(Q)
`Element`(B)	`Top`(S)	`Head`(Q)
`Remove`(B)	`Pop`(S)	`Dequeue`(Q)

3.4 Durchsuchen von Graphen

Wir geben nun für die Suche in einem Graphen bzw. Digraphen $G = (V, E)$ einen Algorithmus `GraphSearch` mit folgenden Eigenschaften an:

`GraphSearch` benutzt eine Prozedur `Explore`, um alle Knoten und Kanten von G zu besuchen.

`Explore`(w) findet Pfade zu allen von w aus erreichbaren Knoten.

Hierzu speichert `Explore`(w) für jeden über eine Kante $\{u, v\}$ bzw. (u, v) neu entdeckten Knoten $v \neq w$ den Knoten u in `parent`(v).

Algorithmus `GraphSearch`(V, E)

```
1  for all v ∈ V, e ∈ E do
2    visited(v) := false
3    parent(v) := nil
4    visited(e) := false
5  for all w ∈ V do
6    if visited(w) = false then Explore(w)
```

Prozedur `Explore`(w)

```
1  visited(w) := true
2  Init(B)
3  Insert(B, w)
4  while ¬Empty(B) do
5    u := Element(B)
6    if ∃ e = {u, v} bzw. e = (u, v) ∈ E : visited(e) = false
       then
7      visited(e) := true
8      if visited(v) = false then
9        visited(v) := true
10       parent(v) := u
11       Insert(B, v)
12   else
13     Remove(B)
```

Satz 36. *Falls der (un)gerichtete Graph G in Adjazenzlisten-Darstellung gegeben ist, durchläuft* `GraphSearch` *alle Knoten und Kanten von G in Zeit $O(|V| + |E|)$.*

Beweis. Offensichtlich wird jeder Knoten u genau einmal zu B hinzugefügt. Dies geschieht zu dem Zeitpunkt, wenn u zum ersten Mal „besucht“ und das Feld `visited` für u auf `true` gesetzt wird. Außerdem werden in Zeile 6 von `Explore` alle von u ausgehenden Kanten durchlaufen, bevor u wieder aus B entfernt wird. Folglich werden tatsächlich alle Knoten und Kanten von G besucht.

Wir bestimmen nun die Laufzeit des Algorithmus `GraphSearch`. Innerhalb von `Explore` wird die while-Schleife für jeden Knoten u genau $(\deg(u) + 1)$-mal bzw. $(\deg^+(u) + 1)$-mal durchlaufen:

- einmal für jeden Nachbarn v von u und
- dann noch einmal, um u aus B zu entfernen.

Insgesamt sind das $\|V\| + 2\|E\|$ im ungerichteten bzw. $\|V\| + \|E\|$ Durchläufe im gerichteten Fall. Bei Verwendung von Adjazenzlisten kann die nächste von einem Knoten v aus noch nicht besuchte Kante e in konstanter Zeit ermittelt werden, falls man für jeden Knoten v einen Zeiger auf (den Endpunkt von) e in der Adjazenzliste von v vorsieht. Die Gesamtlaufzeit des Algorithmus `GraphSearch` beträgt somit $O(\|V\| + \|E\|)$. □

Als nächstes zeigen wir, dass `Explore`(w) zu allen von w aus erreichbaren Knoten v einen (gerichteten) w-v-Pfad liefert. Dieser lässt sich mittels `parent` wie folgt zurückverfolgen. Sei

$$u_i = \begin{cases} v, & i = 0, \\ \texttt{parent}(u_{i-1}), & i > 0 \text{ und } u_{i-1} \neq \texttt{nil} \end{cases}$$

und sei $\ell = \min\{i \geq 0 \mid u_{i+1} = \texttt{nil}\}$. Dann ist $u_\ell = w$ und $p = (u_\ell, \ldots, u_0)$ ein w-v-Pfad. Wir nennen p den ***`parent`*-Pfad** von w zu v.

Satz 37. *Falls beim Aufruf von* `Explore` *alle Knoten und Kanten als unbesucht markiert sind, berechnet* `Explore`(w) *einen (gerichteten)* `parent`*-Pfad von w zu allen erreichbaren Knoten.*

Beweis. Wir zeigen zuerst, dass `Explore`(w) alle von w aus erreichbaren Knoten markiert. Hierzu führen wir Induktion über die Länge ℓ eines kürzesten w-v-Weges.

$\ell = 0$: In diesem Fall ist $v = w$ und w wird in Zeile 1 markiert.

$\ell \rightsquigarrow \ell + 1$**:** Sei v ein Knoten mit Abstand $l + 1$ von w. Dann hat ein Nachbarknoten $u \in N(v)$ den Abstand ℓ von w. Folglich wird u nach IV als besucht markiert. Da u nicht vorher aus B entfernt wird, bis alle seine Nachbarn (bzw. Nachfolger) markiert sind, wird auch v markiert.

Es bleibt zu zeigen, dass `parent` einen Pfad von w zu jedem markierten Knoten v liefert. Hierzu führen wir Induktion über die Anzahl k der vor v markierten Knoten.

$k = 0$**:** In diesem Fall ist $v = w$. Da $\texttt{parent}(w) = \texttt{nil}$ ist, liefert `parent` einen w-v-Pfad (der Länge 0).

$k - 1 \rightsquigarrow k$**:** Sei $u = \texttt{parent}(v)$. Da u vor v markiert wird, liefert `parent` nach IV einen w-u-Pfad. Wegen $u = \texttt{parent}(v)$ ist u der Entdecker von v und daher mit v durch eine Kante verbunden. Somit liefert `parent` auch für v einen w-v-Pfad. □

3.4.1 Berechnung der Zusammenhangskomponenten

Da `Explore`(w) alle von w aus erreichbaren Knoten findet, lassen sich die Zusammenhangskomponenten eines (ungerichteten) Graphen durch folgende Variante von `GraphSearch` bestimmen.

Algorithmus `CC`(V, E)

```
1  k := 0
2  for all v ∈ V, e ∈ E do
3     cc(v) := 0
4     cc(e) := 0
5  for all w ∈ V do
6     if cc(w) = 0 then
7        k := k + 1
8        ComputeCC(k, w)
```

Prozedur `ComputeCC`(k, w)

```
1  cc(w) := k
2  Init(B)
3  Insert(B, w)
4  while ¬Empty(B) do
5     u := Element(B)
6     if ∃ e = {u, v} ∈ E : cc(e) = 0 then
7        cc(e) := k
8        if cc(v) = 0 then
9           cc(v) := k
10          Insert(B, v)
11    else
12       Remove(B)
```

Korollar 38. *Der Algorithmus* `CC`(V, E) *bestimmt für einen Graphen* $G = (V, E)$ *in Linearzeit* $O(\|V\| + \|E\|)$ *sämtliche Zusammenhangskomponenten* $G_k = (V_k, E_k)$ *von* G*, wobei* $V_k = \{v \in V \mid \texttt{cc}(v) = k\}$ *und* $E_k = \{e \in E \mid \texttt{cc}(e) = k\}$ *ist.*

3.4.2 Spannbäume und Spannwälder

In diesem Abschnitt zeigen wir, dass der Algorithmus `GraphSearch` für jede Zusammenhangskomponente eines (ungerichteten) Graphen G einen Spannbaum berechnet.

Definition 39. *Sei* $G = (V, E)$ *ein Graph und* $H = (U, F)$ *ein Untergraph.*

- H *heißt* spannend*, falls* $U = V$ *ist.*
- H *ist ein* spannender Baum *(oder* Spannbaum*) von* G*, falls* H *ein Baum und* $U = V$ *ist.*
- H *ist ein* spannender Wald *(oder* Spannwald*) von* G*, falls* H *ein Wald und* $U = V$ *ist.*

Es ist leicht zu sehen, dass für G genau dann ein Spannbaum existiert, wenn G zusammenhängend ist. Allgemeiner gilt, dass die Spannbäume für die Zusammenhangskomponenten von G einen Spannwald bilden. Tatsächlich berechnet der Algorithmus `GraphSearch` einen solchen Spannwald W, da er für jeden neu entdeckten Knoten v seinen *Entdecker* u im Feld $\texttt{parent}(v)$ speichert.

Betrachte den durch $\texttt{SearchGraph}(V, E)$ erzeugten Graphen $W = (V, E_{\texttt{parent}})$ mit

$$E_{\texttt{parent}} = \Big\{\{\texttt{parent}(v), v\} \mid v \in V \text{ und } \texttt{parent}(v) \neq \texttt{nil}\Big\}.$$

Da $\texttt{parent}(v)$ vor v markiert wird, ist klar, dass W kreisfrei und somit tatsächlich ein Wald ist. Wir bezeichnen W auch als *Suchwald* von G. Kennzeichnen wir in jedem Baum T von W den eindeutig bestimmten Knoten w mit $\texttt{parent}(w) = \texttt{nil}$ als Wurzel von T, so erhalten die Kanten von W hierdurch die Richtung $(\texttt{parent}(v), v)$. W hängt zum einen davon ab, wie die Datenstruktur B implementiert ist (z.B. als Keller oder als Warteschlange). Zum anderen hängt W aber auch von der Reihenfolge der Knoten in den Adjazenzlisten ab.

Da $\texttt{Explore}(w)$ alle von w aus erreichbaren Knoten findet, spannt jeder Baum des Suchwalds W eine Zusammenhangskomponente von G, d.h. die Bäume des Suchwalds sind Spannbäume der Zusammenhangskomponenten.

Korollar 40. *Sei G ein (ungerichteter) Graph.*

- *Der Algorithmus $\texttt{GraphSearch}(V, E)$ berechnet in Linearzeit einen Spannwald W, dessen Bäume die Zusammenhangskomponenten von G spannen.*
- *Falls G zusammenhängend ist, ist W ein Spannbaum für G.*

3.4.3 Breiten- und Tiefensuche

Wie wir gesehen haben, findet $\texttt{Explore}(w)$ sowohl in Graphen als auch in Digraphen alle von w aus erreichbaren Knoten. Als nächstes zeigen wir, dass $\texttt{Explore}(w)$ zu allen von w aus erreichbaren Knoten sogar einen kürzesten Weg findet, falls wir die Datenstruktur B als Warteschlange Q implementieren.

Die Benutzung einer Warteschlange Q zur Speicherung der bereits entdeckten, aber noch nicht abgearbeiteten Knoten bewirkt, dass zuerst alle Nachbarknoten $u_1, \ldots, u_k$ des aktuellen Knotens u besucht werden, bevor ein anderer Knoten aktueller Knoten wird. Da die Suche also zuerst in die Breite geht, spricht man von einer *Breitensuche* (kurz *BFS*, engl. *breadth first search*). Den hierbei berechneten Suchwald bezeichnen wir als *Breitensuchwald*.

Bei Benutzung eines Kellers wird dagegen u_1 aktueller Knoten, bevor die übrigen Nachbarknoten von u besucht werden. Daher führt die Benutzung eines Kellers zu einer *Tiefensuche* (kurz *DFS*, engl. *depth first search*). Der berechnete Suchwald heißt dann *Tiefensuchwald*.

Die Breitensuche eignet sich eher für Distanzprobleme wie z.B. das Finden

- kürzester Wege in Graphen und Digraphen,
- längster Wege in Bäumen (siehe Übungen) oder
- kürzester Wege in Distanzgraphen (Dijkstra-Algorithmus).

Dagegen liefert die Tiefensuche interessante Strukturinformationen wie z.B.

- die zweifachen Zusammenhangskomponenten in Graphen,
- die starken Zusammenhangskomponenten in Digraphen oder
- eine topologische Sortierung bei azyklischen Digraphen (s. Übungen).

Wir betrachten zuerst den Breitensuchalgorithmus.

Algorithmus $\texttt{BFS}(V, E)$

```
1  for all v ∈ V, e ∈ E do
2    visited(v) := false
3    parent(v) := nil
```

```
4     visited(e) := false
5   for all w ∈ V do
6     if visited(w) = false then BFS-Explore(w)
```

Prozedur `BFS-Explore`(w)

```
1   visited(w) := true
2   Init(Q)
3   Enqueue(Q, w)
4   while ¬Empty(Q) do
5     u := Head(Q)
6     if ∃ e = {u, v} bzw. e = (u, v) ∈ E : visited(e) = false
        then
7       visited(e) := true
8       if visited(v) = false then
9         visited(v) := true
10        parent(v) := u
11        Enqueue(Q, v)
12    else
13      Dequeue(Q)
```

Satz 41. *Sei G ein Graph oder Digraph und sei w Wurzel des von* `BFS-Explore`(w) *berechneten Suchbaumes T. Dann liefert* `parent` *für jeden Knoten v in T einen kürzesten w-v-Weg.*

Beweis. Wir führen Induktion über die kürzeste Weglänge ℓ von w nach v in G.

$\ell = 0$: Dann ist $v = w$ und `parent` liefert einen Weg der Länge 0.

$\ell \rightsquigarrow \ell + 1$: Sei v ein Knoten, der den Abstand $\ell + 1$ von w in G hat. Dann existiert ein Knoten $u \in N^-(v)$ (bzw. $u \in N(v)$), der den Abstand ℓ von w in G hat. Nach IV liefert also `parent` einen w-u-Weg der Länge ℓ. Da u erst aus Q entfernt wird, nachdem alle Nachfolger von u entdeckt sind, wird v von u oder einem bereits zuvor in Q eingefügten Knoten z entdeckt. Da Q als Schlange organisiert ist, liefert `parent` von w zu z keinen längeren Weg als von w zu u. Daher folgt in beiden Fällen, dass `parent` einen w-v-Weg der Länge $\ell + 1$ liefert. □

Wir werden später noch eine Modifikation der Breitensuche kennen lernen, die kürzeste Wege in Graphen mit nichtnegativen Kantenlängen findet (Algorithmus von Dijkstra). Als nächstes betrachten wir den Tiefensuchalgorithmus.

Algorithmus `DFS`(V, E)

```
1   for all v ∈ V,  e ∈ E do
2     visited(v) := false
3     parent(v) := nil
4     visited(e) := false
5   for all w ∈ V do
6     if visited(w) = false then DFS-Explore(w)
```

Prozedur `DFS-Explore`(w)

```
1   visited(w) := true
2   Init(S)
3   Push(S, w)
4   while ¬Empty(S) do
5     u := Head(S)
6     if ∃ e = {u, v} bzw. e = (u, v) ∈ E : visited(e) = false
        then
7       visited(e) := true
8       if visited(v) = false then
9         visited(v) := true
10        parent(v) := u
11        Push(S, v)
12    else
13      Pop(S)
```

Die Tiefensuche lässt sich auch rekursiv implementieren. Dies hat den Vorteil, dass kein (expliziter) Keller benötigt wird.

Prozedur `DFS-Explore-rec`(w)

```
1  visited(w) := true
2  while
       ∃ e = {u, v} bzw. e = (u, v) ∈ E : visited(e) = false
       do
3    visited(e) := true
4    if visited(v) = false then
5      parent(v) := w
6      DFS-Explore-rec(v)
```

Da `DFS-Explore-rec`(w) zu `parent`(w) zurückspringt, kann auch das Feld `parent`(w) als Keller fungieren. Daher lässt sich die Prozedur auch nicht-rekursiv ohne zusätzlichen Keller implementieren, indem die Rücksprünge explizit innerhalb einer Schleife ausgeführt werden.

Die Kanten eines Graphen $G = (V, E)$ werden durch den Tiefensuchwald $W = (V, E_{\texttt{parent}})$ in zwei Klassen eingeteilt. Die Kanten $\{\texttt{parent}(v), v\} \in E_{\texttt{parent}}$ heißen *Baumkanten*. Daneben gibt es noch *Rückwärtskanten* $\{u, v\}$, die einen Knoten u mit einem Knoten v auf dem `parent`-Pfad von u verbinden.

So genannte *Querkanten*, die zwei Knoten in W verbinden, so dass keiner auf dem `parent`-Pfad des anderen liegt, kann es dagegen nicht geben. Sonst müsste nämlich einer von beiden abgearbeitet sein, bevor der andere entdeckt wird. Dies ist jedoch bei einer Tiefensuche nicht möglich, da kein Knoten aus dem Keller entfernt wird, bevor nicht alle von ihm aus erreichbaren Knoten entdeckt sind.

Das Fehlen von Querkanten spielt bei manchen Anwendungen eine wichtige Rolle, etwa bei der Zerlegung eines Graphen G in seine *zweifachen Zusammenhangskomponenten*.

Als nächstes betrachten wir die Tiefensuche auf Digraphen.

Definition 42. *Sei $G = (V, E)$ ein Digraph.*

- *Ein Knoten $w \in V$ heißt* Wurzel *von G, falls alle Knoten $v \in V$ von w aus erreichbar sind (d.h. es gibt einen gerichteten w-v-Weg in G).*
- *G heißt* gerichteter Baum*, wenn G eine Wurzel hat, kreisfrei ist und jeder Knoten $v \in V$ Eingangsgrad $\deg^-(v) \leq 1$ hat.*
- *G heißt* gerichteter Wald*, wenn G kreisfrei ist und jeder Knoten $v \in V$ Eingangsgrad $\deg^-(v) \leq 1$ hat.*
- *Ein Knoten $v \in V$ heißt* Blatt *von G, falls v den Ausgangsgrad $\deg^-(v) = 0$ hat.*

In einem gerichteten Baum liegen die Kantenrichtungen durch die Wahl der Wurzel bereits eindeutig fest. Daher kann bei bekannter Wurzel auf die Angabe der Kantenrichtungen verzichtet werden. Man spricht dann auch von einem *Wurzelbaum*.

Anders als im Fall eines ungerichteten Eingabegraphen, bei dem die Baumkanten durch die Kennzeichnung des Startknotens als Wurzel nur implizit gerichtet sind, geben wir den Baumkanten des Suchwalds eines Digraphen eine explizite Richtung $(\texttt{parent}(v), v)$. Daher erhalten wir in diesem Fall einen gerichteten Suchwald W.

Definition 43. *Sei $W = (V, E)$ ein gerichteter Wald und $v \in V$ ein beliebiger Knoten.*

- *Ein Knoten $u \in V$ heißt* Nachfahre *von v, falls in W ein gerichteter v-u-Weg existiert.*
- *Wir bezeichnen dann v auch als* Vorfahren *von u in W.*
- *Gilt zudem $u \neq v$, so sprechen wir auch von* echten *Nach- bzw. Vorfahren.*
- *Den in W durch alle Nachfahren von v induzierten Baum mit der Wurzel v bezeichnen wir mit $T_W(v)$ bzw. einfach mit $T(v)$, falls W aus dem Kontext ersichtlich ist.*

Die Kanten eines Digraphen werden durch eine Tiefensuche in vier Klassen eingeteilt. Die Kanten $(\texttt{parent}(v), v)$ des Tiefensuchwaldes W heißen wieder *Baumkanten*. Eine *Rückwärtskante* (u, v) verbindet einen Knoten u mit einem echten Vorfahren v von u in W. Dieser liegt auf dem `parent`-Pfad zu u. Eine *Vorwärtskante* (v, u) verbindet einen Knoten v mit einem Nachfahren u von v in W (hier kann $u = v$ sein). Alle anderen Kanten heißen *Querkanten*.

Der Typ jeder Kante lässt sich algorithmisch leicht bestimmen, wenn wir bei der Tiefensuche noch folgende Zusatzinformationen speichern.

- Ein neu entdeckter Knoten wird grau gefärbt. Wenn er abgearbeitet ist, wird er schwarz. Zu Beginn sind alle Knoten weiß.
- Zudem merken wir uns die Reihenfolge, in der die Knoten entdeckt werden, in einem Feld `k`.

Die folgende Variante von `DFS` berechnet diese Informationen.

Algorithmus `DFS`(V, E)

```
1  k := 0
2  for all v ∈ V, e ∈ E do
3     farbe(v) := weiß
4     visited(e) := false
5  for all u ∈ V do
6     if farbe(u) = weiß then DFS-Explore(u)
```

Prozedur `DFS-Explore`(u)

```
1  farbe(u) := grau
2  k := k + 1
3  k(u) := k
4  while ∃ e = (u, v) ∈ E : visited(e) = false do
5     visited(e) := true
6     if farbe(v) = weiß then
7        DFS-Explore(v)
8  farbe(u) := schwarz
```


Nun lässt sich der Typ jeder Kante $e = (u, v)$ bei ihrem Besuch in Zeile 6 anhand der Werte von $\texttt{farbe}(v)$ und $\texttt{k}(v)$ wie folgt bestimmen:

Baumkante: $\texttt{farbe}(v) =$ weiß,
Vorwärtskante: $\texttt{farbe}(v) \neq$ weiß und $\texttt{k}(v) \geq \texttt{k}(u)$,
Rückwärtskante: $\texttt{farbe}(v) =$ grau und $\texttt{k}(v) < \texttt{k}(u)$,
Querkante: $\texttt{farbe}(v) =$ schwarz und $\texttt{k}(v) < \texttt{k}(u)$.

3.4.4 Starke Zusammenhangskomponenten

Sei $G = (V, E)$ ein Digraph. Dann ist leicht zu sehen, dass die Relation

$$S = \{(u, v) \in V \times V \mid \text{es gibt in } G \text{ einen } u\text{-}v\text{-Weg und einen } v\text{-}u\text{-Weg}\}$$

eine Äquivalenzrelation ist. Für $(u, v) \in S$ schreiben wir auch kurz $u \sim v$.

Definition 44. *Die durch die Äquivalenzklassen $U_1, \ldots, U_k$ von S induzierten Teilgraphen $G[U_1], \ldots, G[U_k]$ heißen die* starken Zusammenhangskomponenten *(engl.* strongly connected components*) von G.*

Satz 45. *Sei $G = (V, E)$ ein Digraph mit den starken Zusammenhangskomponenten $G[U_1], \ldots, G[U_k]$. Dann ist der Digraph (C, D) mit $C = \{1, \ldots, k\}$ und*

$$D = \{(i, j) \mid 1 \leq i \neq j \leq k \wedge \exists u \in U_i, v \in U_j : (u, v) \in E\}$$

azyklisch.

Beweis. Da der Digraph (C, D) schlingenfrei ist, müsste ein Zyklus mindestens zwei verschiedene Knoten $i \neq j$ enthalten. Dann wären aber alle Knoten in den beiden Komponenten $G[U_i]$ und $G[U_j]$ gegenseitig erreichbar, d.h. alle Knoten in $U_i \cup U_j$ müssten in derselben Komponente liegen (Widerspruch). □

Sei G ein Digraph mit den starken Zusammenhangskomponenten $G[U_1], \ldots, G[U_k]$. Führen wir auf G eine Tiefensuche durch, so erhalten wir einen Tiefensuchwald W. Für einen Knoten v bezeichne $T(v)$ den in W durch alle Nachfahren von v induzierten Baum mit Wurzel v. Für $i = 1, \ldots, k$ sei s_i der erste bei der Tiefensuche innerhalb von U_i besuchte Knoten. Wir nennen s_i den *Startknoten* von U_i. Da s_i erst schwarz wird, nachdem alle von s_i aus erreichbaren Knoten besucht wurden, ist U_i komplett im Unterbaum $T(s_i)$ enthalten.

Weiter sei V_i die Menge derjenigen Knoten in $T(s_i)$, die zwar Nachfahre von s_i sind, aber nicht gleichzeitig Nachfahre von einem in $T(s_i)$ enthaltenen Startknoten $s_j \neq s_i$. Wir behaupten, dass $U_i \subseteq V_i$ gilt. Da die Mengen U_i alle Knoten überdecken, impliziert dies $U_i = V_i$ für alle $i = 1, \ldots, k$. Nehmen wir also an, es gäbe einen Knoten $u \in U_i$, der nicht in V_i enthalten ist. Dann ist u zu einem Startknoten $s_j \neq s_i$ äquivalent, muss also sowohl in $T(s_i)$ (wegen $u \in U_i$) als auch in $T(s_j)$ (wegen $u \sim s_j$) liegen. Folglich muss s_j entweder Nach- oder Vorfahre von s_i sein. Da u zu U_i gehört, kann aber s_j kein Nachfahre von s_i sein. s_j kann aber auch kein Vorfahre von s_i sein, da andernfalls wegen $u \sim s_j$ und $u \in T(s_i)$ sofort $u \sim s_i$ folgen würde (Widerspruch).

Die Mengen U_i lassen sich also leicht bestimmen, falls wir die Startknoten s_i während der Tiefensuche identifizieren können. Hierzu betrachten wir für $u \in U_i$ die Funktion

$$\mathtt{low}(u) := \min\{\mathtt{k}(v) \mid v \in T(s_i) \text{ und es gibt einen } u\text{-}v\text{-Weg in } G\}.$$

Da alle Knoten in $T(s_i)$ nach s_i entdeckt werden, ist $\mathtt{k}(v) \geq \mathtt{k}(s_i)$ für alle $v \in T(s_i)$ und daher folgt $\mathtt{low}(u) \geq \mathtt{k}(s_i)$. Zudem existiert wegen $u \sim s_i$ ein u-s_i-Weg und somit ist $\mathtt{low}(u) = \mathtt{k}(s_i)$. Folglich erfüllen genau die Startknoten $u \in \{s_1, \ldots, s_k\}$ die Eigenschaft $\mathtt{low}(u) = \mathtt{k}(u)$.

Um die Knoten s_i bei der Tiefensuche effizient als Startknoten identifizieren zu können, betrachten wir für $u \in U_i$ die Funktion

$$\mathtt{l}(u) := \min\{\mathtt{k}(v) \mid v = u \vee \exists u' \in T(u), v \in T(s_i) : (u', v) \in E\}.$$

Dann gilt $\mathtt{k}(s_i) \leq \mathtt{l}(u) \leq \mathtt{k}(u)$, wobei $\mathtt{l}(u) = \mathtt{k}(u)$ nur für $u = s_i$ gilt. Ist nämlich $u \neq s_i$, so führt jeder u-s_i-Weg aus $T(u)$ heraus. Dies kann jedoch nur mittels einer (Rückwärts- oder Quer-) Kante (u', v) mit $u' \in T(u)$, $v \in T(s_i)$ und $\mathtt{k}(v) < \mathtt{k}(u)$ geschehen.

Der folgende Algorithmus `SCC` berechnet für jeden Knoten u den Wert $\mathtt{l}(u)$ und gibt der Reihe nach die Mengen U_i aus. Dass `SCC` bis zum letzten Besuch eines Knotens u tatsächlich den Wert $\mathtt{l}(u)$ korrekt berechnet, lässt sich leicht induktiv über die Anzahl der zuvor abgearbeiteten Knoten zeigen. `SCC` speichert alle entdeckten Knoten, die noch keiner Menge U_i zugeordnet werden konnten, in einem Keller S. Das Feld `onStack` speichert die Information, welche Knoten sich aktuell in S befinden.

Besitzt ein Knoten u bei seinem letzten Besuch den Wert $\mathtt{l}(u) = \mathtt{k}(u)$, so wird die Prozedur `Output-SCC`(u) aufgerufen. `Output-SCC`(u) leert den Keller S bis einschließlich u und gibt diese Knoten als neu entdeckte Menge U_i aus.

Algorithmus `SCC`(V, E)

```
1  k := 0
2  Init(S)
3  for all v ∈ V, e ∈ E do
4    visited(e) := false
5    k(v) := 0
6    onStack(v) := false
7  for all u ∈ V do
8    if k(u) = 0 then Compute-SCC(u)
```

Prozedur `Compute-SCC`(u)

```
1  k := k + 1
2  k(u) := k
3  l(u) := k
4  Push(S, u)
5  while ∃ e = (u, v) ∈ E : visited(e) = false do
```

```
6     visited(e) := true
7     if k(v) = 0 then
8       Compute-SCC(v)
9       l(u) := min{l(u), l(v)}
10    else if onStack(v) = true then
11      l(u) := min{l(u), k(v)}
12  if l(u) = k(u) then
13    Output-SCC(u)
```

Prozedur `Output-SCC`(u)

```
1  write(Neue Komponente: )
2  repeat
3    v := Pop(S)
4    onStack(v) := false
5    write(v)
6  until(v = u)
```

3.5 Kürzeste Pfade in Distanzgraphen

In vielen Anwendungen tritt das Problem auf, einen kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem Digraphen zu finden, dessen Kanten (u, v) eine vorgegebene *Länge* $l(u, v)$ haben. Die Länge eines Weges $p = (v_0, \ldots, v_\ell)$ ist

$$l(p) = \sum_{i=0}^{\ell-1} l(v_i, v_{i+1}).$$

Die kürzeste Pfadlänge von s nach t wird als *Distanz* $d(s, t)$ von s zu t bezeichnet,

$$d(s, t) = \min\{l(p) \mid p \text{ ist ein } s\text{-}t\text{-Weg}\}.$$

Falls kein s-t-Weg existiert, setzen wir $d(s, t) = \infty$. In vielen Fällen haben alle Kanten in E eine nichtnegative Länge $l(u, v) \geq 0$. Dann wird $D = (V, E, l)$ auch *Distanzgraph* genannt.

3.5.1 Der Dijkstra-Algorithmus

Für die Suche in Distanzgraphen ist es vorteilhaft, die Knoten v, zu denen bereits ein s-v-Weg gefunden wurde, dessen Optimalität aber noch nicht garantiert werden kann, in einer Menge P zu speichern. Damit die Suche effizient ist, sollte die Datenstruktur für P folgende Operationen effizient implementieren.

`Init`(P)**:** Initialisiert P als leere Menge.

`Update`(P, u, d)**:** Erniedrigt den Wert von u auf d (nur wenn der aktuelle Wert größer als d ist). Ist u noch nicht in P enthalten, wird u mit dem Wert d in P eingefügt.

`RemoveMin`(P)**:** Gibt ein Element aus P mit dem kleinsten Wert zurück und entfernt es aus P. Ist P leer, wird `nil` zurückgegeben.

Der Dijkstra-Algorithmus findet einen kürzesten Weg vom Startknoten s zu allen erreichbaren Knoten (single-source shortest-path problem). Hierzu führt der Algorithmus eine modifizierte Breitensuche mit dem Startknoten s aus. Wird nur ein kürzester Weg von s zu einem bestimmten Zielknoten t gesucht, kann man die Suche auch abbrechen, sobald t als fertig markiert ist.

Voraussetzung für die Korrektheit des Algorithmus' ist, dass alle Kanten in E eine nichtnegative Länge $l(u, v) \geq 0$ haben.

Algorithmus `Dijkstra`(V, E, l, s)

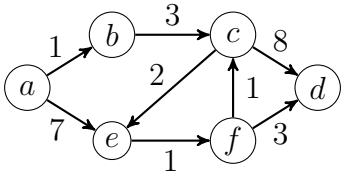
```
1  for all v ∈ V do
2    g(v) := ∞
3    parent(v) := nil
4    done(v) := false
```

```
5  g(s) := 0
6  Init(P)
7  Update(P, s, 0)
8  while u := RemoveMin(P) ≠ nil do
9    done(u) := true
10   for all v ∈ N⁺(u) do
11     if done(v) = false ∧ g(u) + l(u, v) < g(v) then
12       g(v) := g(u) + l(u, v)
13       Update(P, v, g(v))
14       parent(v) := u
```

Der Algorithmus führt eine Breitensuche durch und speichert den aktuellen Breitensuchbaum T im Feld `parent`. Das Feld `g` dient zur Speicherung der aktuellen Wurzeldistanzen im Breitensuchbaum T. Knoten außerhalb von T haben den Wert ∞.

In jedem Schleifendurchlauf wird in Zeile 8 ein unfertiger Knoten u in T mit kleinstem `g`-Wert gewählt und als fertig markiert. Anschließend werden alle unfertigen Nachfolger v von u an den Knoten u in T an- bzw. umgehängt, falls sich dadurch ihre Wurzeldistanz $\mathtt{g}(v)$ verkleinert.

Beispiel 46. *Betrachte nebenstehenden Distanzgraphen mit dem Startknoten a. Dann führt der Dijkstra-Algorithmus die folgenden Rechenschritte aus.*



gewählt	besuchte Kanten	Update-Operationen	Inhalt von P
$(a, 0)$	$(a, b), (a, e)$	$(b, 1), (e, 7)$	$(b, 1), (e, 7)$
$(b, 1)$	(b, c)	$(c, 4)$	$(c, 4), (e, 7)$
$(c, 4)$	$(c, d), (c, e)$	$(d, 12), (e, 6)$	$(e, 6), (d, 12)$
$(e, 6)$	(e, f)	$(f, 7)$	$(f, 7), (d, 12)$
$(f, 7)$	$(f, c), (f, d)$	$(d, 10)$	$(d, 10)$
$(d, 10)$	–	–	–

◁

Als nächstes beweisen wir die Korrektheit des Dijkstra-Algorithmus'.

Satz 47. *Sei $D = (V, E, l)$ ein Distanzgraph und sei $s \in V$. Dann berechnet `Dijkstra`(V, E, l, s) für alle von s aus erreichbaren Knoten $t \in V$ einen kürzesten s-t-Weg. Dieser Weg lässt sich mittels `parent` von t zu s zurückverfolgen.*

Beweis. Wir zeigen zuerst, dass alle von s aus erreichbaren Knoten $t \in V$ als fertig markiert werden. Sei T der durch `parent` beschriebene Breitensuchbaum bei Abbruch der while-Schleife. Dann werden genau die Knoten in T als fertig markiert. Zudem werden für jeden Knoten u in T spätestens dann, wenn u in Zeile 8 gewählt wird, sämtliche Nachfolger von u zu T hinzugefügt. Also enthält T alle von s aus erreichbaren Knoten.

Als nächstes zeigen wir, dass `parent` für jeden Knoten t in T einen kürzesten s-t-Weg liefert, d.h. es gilt $\mathtt{g}(t) \leq d(s, t)$ für alle Knoten t in T. Wir führen Induktion über die Anzahl k der vor t gewählten Knoten.

$k = 0$**:** In diesem Fall ist $t = s$ und `parent` liefert einen kürzesten s-t-Weg (der Länge 0).

$k - 1 \rightsquigarrow k$**:** Sei $s = v_0, \ldots, v_\ell = t$ ein *kürzester* s-t-Weg in G und sei v_i der Knoten mit maximalem Index i auf diesem Weg, der vor t gewählt wird.

Nach IV gilt dann

$$d(s, v_i) = \mathtt{g}(v_i). \tag{3.1}$$

Zudem ist

$$\mathtt{g}(v_{i+1}) \leq \mathtt{g}(v_i) + l(v_i, v_{i+1}). \tag{3.2}$$

Da t im Fall $t \neq v_{i+1}$ vor v_{i+1} gewählt wird, ist

$$\mathtt{g}(t) \leq \mathtt{g}(v_{i+1}). \tag{3.3}$$

Daher folgt

$$\begin{aligned}\texttt{g}(t) &\overset{(3.3)}{\leq} \texttt{g}(v_{i+1}) \overset{(3.2)}{\leq} \texttt{g}(v_i) + l(v_i, v_{i+1}) \overset{(3.1)}{=} d(s, v_i) + l(v_i, v_{i+1}) \\ &= d(s, v_{i+1}) \leq d(s, t).\end{aligned}$$

□

Um die Laufzeit des Dijkstra-Algorithmus' abzuschätzen, überlegen wir uns zuerst, wie oft die einzelnen Operationen auf der Datenstruktur P ausgeführt werden. Sei $n = \|V\|$ die Anzahl der Knoten und $m = \|E\|$ die Anzahl der Kanten des Eingabegraphen.

- Die `Init`-Operation wird nur einmal ausgeführt.
- Da die while-Schleife für jeden Knoten höchstens einmal durchlaufen wird, wird die `RemoveMin`-Operation höchstens n-mal ausgeführt.
- Wie die Prozedur `BFS-Explore` besucht der Dijkstra-Algorithmus jede Kante maximal einmal. Daher wird die `Update`-Operation höchstens m-mal ausgeführt.

Beobachtung 48. *Bezeichne $In(n)$, $RM(n)$ und $Up(n)$ den Aufwand zum Ausführen der Operationen `Init`, `RemoveMin` und `Update` für den Fall, dass P nicht mehr als n Elemente aufzunehmen hat. Dann ist die Laufzeit des Dijkstra-Algorithmus' durch*

$$O(n + m + In(n) + n \cdot RM(n) + m \cdot Up(n))$$

beschränkt.

Die Laufzeit hängt also stark davon ab, wie wir die Datenstruktur P implementieren. Falls alle Kanten die gleiche Länge haben, wachsen die Distanzwerte der Knoten monoton in der Reihenfolge ihres (ersten) Besuchs. D.h. wir können P als Warteschlange implementieren. Dies führt wie bei der Prozedur `BFS-Explore` auf eine Laufzeit von $O(n + m)$.
Für den allgemeinen Fall, dass die Kanten unterschiedliche Längen haben, betrachten wir folgende drei Möglichkeiten.

1. Da die Felder `g` und `done` bereits alle nötigen Informationen enthalten, kann man auf die (explizite) Implementierung von P auch verzichten. In diesem Fall kostet die `RemoveMin`-Operation allerdings Zeit $O(n)$, was auf eine Gesamtlaufzeit von $O(n^2)$ führt.

   Dies ist asymptotisch optimal, wenn G relativ dicht ist, also $m = \Omega(n^2)$ Kanten enthält. Ist G dagegen relativ dünn, d.h. $m = o(n^2)$, so empfiehlt es sich, P als Prioritätswarteschlange zu implementieren.

2. Es ist naheliegend, P in Form eines Heaps H zu implementieren. In diesem Fall lassen sich die `Update`- und `RemoveMin`-Operationen in Zeit $O(\log n)$ implementieren.

   Dies setzt allerdings voraus, dass wir beim Ausführen der `Update`-Operation nicht erst in H den Knoten v suchen müssen, dessen Wert erniedrigt werden soll. Die Suche nach v in H lässt sich vermeiden, indem wir die `Update`-Operation durch eine `Insert`-Operation ersetzen. Dies führt zwar dazu, dass der gleiche Knoten evtl. mehrmals mit unterschiedlichen Werten in H gespeichert wird.

   Die Korrektheit bleibt aber dennoch erhalten, wenn wir nur die erste Entnahme eines Knotens aus H beachten und die übrigen ignorieren.

   Da für jede Kante höchstens ein Knoten in H eingefügt wird, erreicht H maximal die Größe n^2 und daher sind die Heap-Operationen immer noch in Zeit $O(\log n^2) = O(\log n)$ ausführbar. Insgesamt erhalten wir somit eine Laufzeit von $O(n + m \log n)$.

   Die Laufzeit von $O(n + m \log n)$ bei Benutzung eines Heaps ist zwar für dünne Graphen sehr gut, aber für dichte Graphen schlechter als die implizite Implementierung von P mithilfe der Felder `g` und `done`.

3. Als weitere Möglichkeit kann P auch in Form eines so genannten *Fibonacci-Heaps* F implementiert werden. Dieser benötigt nur eine konstante amortisierte Laufzeit $O(1)$ für die `Update`-Operation und $O(\log n)$ für die `RemoveMin`-Operation. Insgesamt führt dies auf eine Laufzeit von $O(m + n \log n)$.

Eine offene Frage ist, ob es auch einen Algorithmus mit Laufzeit $O(n + m)$ gibt. Wir fassen die verschiedenen Möglichkeiten zur Implementation der Datenstruktur P in folgender Tabelle zusammen.

	implizit	Heap	Fibonacci-Heap
`Init`	$O(1)$	$O(1)$	$O(1)$
`Update`	$O(1)$	$O(\log n)$	$O(1)$
`RemoveMin`	$O(n)$	$O(\log n)$	$O(\log n)$
Gesamtlaufzeit	$O(n^2)$	$O(n + m \log n)$	$O(m + n \log n)$

3.6 Negative Kantengewichte

In manchen Anwendungen treten auch negative Kantengewichte auf. Gesucht ist dann meist ein kürzester Pfad (also ein kürzester Weg, der jeden Knoten höchstens einmal besucht). Will man beispielsweise einen längsten Pfad in einem Distanzgraphen bestimmen, so kann man dies leicht auf die Bestimmung eines kürzesten Pfades in einem Graphen mit negativen Kantengewichten zurückführen, indem man alle Kantengewichte mit -1 multipliziert.

Die Komplexität des Problems hängt wesentlich davon ab, ob man gerichtete Kreise mit negativer Länge zulässt oder nicht. Falls solche Kreise zulässig sind, ist das Problem NP-hart. Andernfalls existieren effiziente Algorithmen wie z.B. der Bellman-Ford-Algorithmus (BF-Algorithmus) oder der Bellman-Ford-Moore-Algorithmus (BFM-Algorithmus). Diese Algorithmen lösen das single-source shortest-path Problem mit einer Laufzeit von $O(nm)$ im schlechtesten Fall.

Der Ford-Algorithmus arbeitet ganz ähnlich wie der Dijkstra-Algorithmus, betrachtet aber jede Kante nicht nur ein- sondern mehrmals. In seiner einfachsten Form wählt der Algorithmus wiederholt eine Kante $e = (u, v)$, so dass sich der aktuelle Distanzwert $g(v)$ auf Grund der Ungleichung

$$\mathtt{g}(u) + \ell(u, v) < \mathtt{g}(v)$$

verkleinern lässt. Ein solcher Schritt wird auch als *Relaxation* von e bezeichnet, da e nun nicht mehr auf eine Veränderung der g-Werte „drängt“. Es ist leicht zu zeigen, dass $g(v)$ für jeden Knoten v den optimalen Wert $g(v) = d(s, v)$ hat, sobald alle Kante relaxiert sind. Es ist klar, dass die Laufzeit stark von der Reihenfolge abhängt, in der die Kanten auf eine Relaxation überprüft werden.

3.6.1 Der Bellman-Ford-Algorithmus

Diese Variante prüft in $n - 1$ Iterationen jeweils alle Kanten auf Relaxation. Sind danach weitere Relaxationen möglich, muss ein negativer Kreis existieren. Die Laufzeit ist offensichtlich $O(nm)$ und die Korrektheit folgt leicht durch Induktion über die minimale Anzahl von Kanten eines kürzesten s-t-Weges. Wird bei jeder Relaxation einer Kante (u, v) der Vorgänger u im Feld $\mathtt{parent}(v)$ vermerkt, so lässt sich ein kürzester Pfad von s zu allen erreichbaren Knoten rekonstruieren (bzw. ein negativer Kreis, falls dieser existiert).

Algorithmus $\mathtt{BF}(V, E, \ell, s)$

```
1  for all v ∈ V do
2     g(v) := ∞
3     parent(v) := nil
4  g(s) := 0
5  for i := 1 to n − 1 do
6     for all (u, v) ∈ E do
7        if g(u) + ℓ(u, v) < g(v) then
```

```
8          g(v) := g(u) + ℓ(u,v)
9          parent(v) := u
10   for all (u,v) ∈ E do
11     if g(u) + ℓ(u,v) < g(v) then
12       error(es gibt einen negativen Kreis)
```

3.6.2 Der Bellman-Ford-Moore-Algorithmus

Die BFM-Variante erzielt im Durchschnittsfall eine deutliche Laufzeit-Verbesserung, da sie im i-ten Durchlauf nur solche Kanten (u, v) prüft, für die der Wert von $g(u)$ im $(i-1)$-ten Durchlauf erniedrigt wurde. Am einfachsten lässt sich die BFM-Strategie unter Benutzung einer Schlange Q zur Speicherung der aktiven Knoten umsetzen.

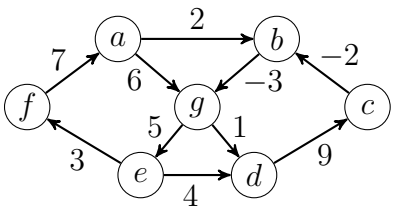
Algorithmus BFM(V, E, ℓ, s)

```
1  for all v ∈ V do
2    g(v) := ∞, parent(v) := nil, inQueue(v) := false
3  g(s) := 0, Init(Q), Enqueue(Q,(0,s)), inQueue(s) := true
4  while (i,u) := Dequeue(Q) ≠ nil and i < n do
5    inQueue(u) := false
6    for all v ∈ N⁺(u) do
7      if g(u) + ℓ(u,v) < g(v) then
8        g(v) := g(u) + ℓ(u,v)
9        parent(v) := u
10       if inQueue(v) = false then
11         Enqueue(Q,(i+1,v))
12         inQueue(v) := true
13 if i = n then
14   error(es gibt einen negativen Kreis)
```

Zu Beginn wird nur der Startknoten s in Q eingefügt. Danach läuft der Algorithmus in *Phasen* ab, wobei die Phasennummer i zusammen mit den Knoten in Q gespeichert werden kann. Werden mehr als $n-1$ Phasen durchlaufen, so bricht der Algorithmus mit der Fehlermeldung ab, dass ein negativer Kreis existiert. Es ist leicht zu sehen, dass der BFM-Algorithmus genau dieselben Relaxationen ausführt wie der BF-Algorithmus und somit ebenfalls korrekt ist.

Für kreisfreie Graphen lässt sich die Laufzeit auf $O(n+m)$ reduzieren, falls die Nachfolger in Zeile 6 in topologischer Sortierung gewählt werden. Dies liegt daran, dass in diesem Fall jeder Knoten höchstens einmal in die Schlange eingefügt wird.

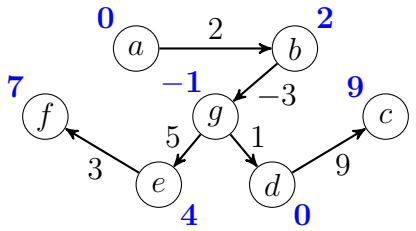
Beispiel 49. *Betrachte untenstehenden kantenbewerteten Digraphen mit dem Startknoten a. Die folgende Tabelle zeigt jeweils den Inhalt der Schlange Q, bevor der BFM-Algorithmus das nächste Paar (i, v) von Q entfernt. Dabei sind neben der Phasennummer i und dem Knoten v auch noch der* parent*-Knoten und der g-Wert von v angegeben, obwohl diese nicht in Q gespeichert werden.*



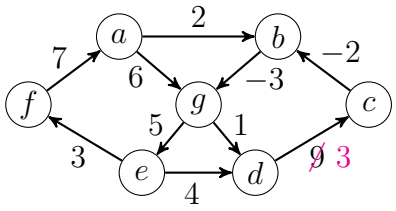
⇑			
$(0,a,\mathtt{nil},0)$			
⇑	$(1,b,a,2)$		
	$(1,g,a,6)$	$(1,g,b,-1)$	
			$(2,d,g,0)$
			$(2,e,g,4)$
$(2,e,g,4)$			
$(3,c,d,9)$	$(3,c,d,9)$		
	$(3,f,e,7)$	$(3,f,e,7)$	

Eintrag $\hat{=}$ *(Phasennr. i, Knoten v,* parent$(v), g(v))$

Die berechneten Entfernungen mit den zugehörigen parent*-Pfaden sind in folgendem Suchbaum wiedergegeben:*



Als nächstes betrachten wir den folgenden Digraphen:

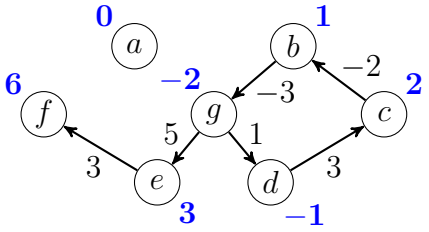


Da dieser einen negativen Kreis enthält, der vom Startknoten aus erreichbar ist, lassen sich die Entfernungen zu manchen Knoten beliebig verkleinern und der Algorithmus bricht erst ab, sobald ein Knoten mit der Phasennummer $i = n = 7$ aus der Schlange Q entnommen wird.

$$
\begin{array}{|c|c|c|c|}
\multicolumn{1}{c}{\Uparrow} & & & \\
(0,a,\texttt{nil},0) & & & \\
\Uparrow & (1,b,a,2) & & \\
 & (1,g,a,6) & (1,g,b,-1) & \\
 & & & (2,d,g,0) \\
 & & & (2,e,g,4) \\
(2,e,g,4) & & & \\
(3,c,d,3) & (3,c,d,3) & & \\
 & (3,f,e,7) & (3,f,e,7) & \\
 & & (4,b,c,1) & (4,b,c,1)
\end{array}
$$

$$
\begin{array}{|c|c|c|c|}
(5,g,b,-2) & & & \\
 & (6,d,g,-1) & & \\
 & (6,e,g,3) & (6,e,g,3) & \\
 & & (7,c,d,2) & (7,c,d,2) \\
 & & & (7,f,e,6)
\end{array}
$$

Da nun der Knoten c mit der Phasennummer $i = n = 7$ aus der Schlange entnommen wird, bricht der Algorithmus an dieser Stelle mit der Meldung ab, dass negative Kreise existieren. Ein solcher Kreis (im Beispiel: c, b, g, d, c) lässt sich bei Bedarf anhand der `parent`*-Funktion aufspüren, indem wir den* `parent`*-Weg zu c zurückverfolgen: c, d, g, b, c.*



◁

3.7 Berechnung von allen kürzesten Wegen

Sei $G = (V, E)$ ein endlicher Digraph mit $V = \{1, \ldots, n\}$. Weiter sei ℓ eine Kantenbewertungsfunktion, die jeder Kante $(i, j) \in E$ eine Länge $\ell(i, j) \in \mathbb{Z}$ zuweist. Wir setzen voraus, dass kein Kreis eine negative Länge hat. Insbesondere muss also für jede Schlinge $(i, i) \in E$ der Wert von $\ell(i, i) \geq 0$ sein.

Für einen Weg $P = (v_1, v_2, \ldots, v_{k-1}, v_k)$ heißen $v_2, \ldots, v_{k-1}$ *innere Knoten* von P. Wir nennen P *nichttrivial*, wenn er mindestens eine Kante durchläuft. Bezeichne $\boldsymbol{d}_k(i, j)$ die Länge eines kürzesten nichttrivialen Weges von i nach j, dessen inneren Knoten alle aus der

Menge $\{1, \ldots, k\}$ stammen. Dann gilt

$$\mathsf{d}_0(i,j) = \begin{cases} \ell(i,j), & (i,j) \in E, \\ \infty, & \text{sonst.} \end{cases}$$

Da wir nur an nichttrivialen Wegen interessiert sind, setzen wir $\mathsf{d}_0(i,i)$ auf ∞, falls die Schlinge (i,i) nicht vorhanden ist. Für den Fall, dass auch triviale Wege berücksichtigt werden sollen, ist $\mathsf{d}_0(i,i)$ dagegen mit dem Wert 0 zu initialisieren. Weiter gilt

$$\mathsf{d}_k(i,j) = \min\Big\{\mathsf{d}_{k-1}(i,j), \mathsf{d}_{k-1}(i,k) + \mathsf{d}_{k-1}(k,j)\Big\}.$$

3.7.1 Der Floyd-Warshall-Algorithmus

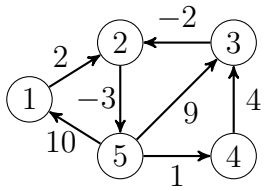
Der Algorithmus von Floyd-Warshall berechnet nichttriviale kürzeste Wege zwischen je zwei Knoten unter der Voraussetzung, dass kein Kreis eine negative Länge hat.

Algorithmus `Floyd-Warshall`(V, E, ℓ)

```
1  for i := 1 to n do
2    for j := 1 to n do
3      if (i,j) ∈ E then d_0(i,j) := ℓ(i,j) else d_0(i,j) := ∞
4  for k := 1 to n do
5    for i := 1 to n do
6      for j := 1 to n do
7        d_k(i,j) = min{d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)}
```

Die Laufzeit ist offenbar $O(n^3)$. Da die d_k-Werte nur von den d_{k-1}-Werten abhängen, ist der Speicherplatzbedarf $O(n^2)$. Die Existenz negativer Kreise lässt sich daran erkennen, dass mindestens ein Diagonalelement $\mathsf{d}_k(i,i)$ einen negativen Wert erhält.

Beispiel 50. *Betrachte folgenden kantenbewerteten Digraphen:*



d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	9	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	9	1	∞

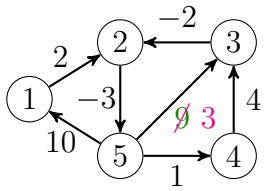
d_2	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	∞	4	∞	∞
5	10	12	9	1	9

d_3	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	2	4	∞	−1
5	10	7	9	1	4

d_4	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	2	4	∞	−1
5	10	3	5	1	0

d_5	1	2	3	4	5
1	9	2	4	0	−1
2	7	0	2	−2	−3
3	5	−2	0	−4	−5
4	9	2	4	0	−1
5	10	3	5	1	0

Als nächstes betrachten wir folgenden Digraphen:



d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	3	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	3	1	∞

d_2	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	∞	4	∞	∞
5	10	12	3	1	9

d_3	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d_4	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d_5	1	2	3	4	5
1	9	0	2	0	-3
2	7	-2	0	-2	-5
3	5	-4	-2	-4	-7
4	9	0	2	0	-3
5	8	-1	1	-1	-4

◁

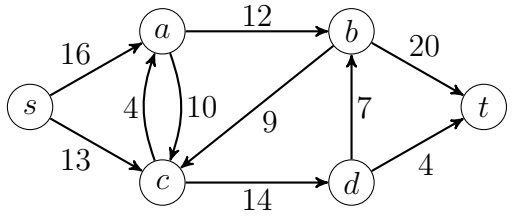
3.8 Flüsse in Netzwerken

Definition 51.

a) *Ein* Netzwerk $N = (V, E, s, t, c)$ *besteht aus einem gerichteten Graphen* $G = (V, E)$ *mit einer* Quelle $s \in V$ *und einer* Senke $t \in V$ *sowie einer* Kapazitätsfunktion $c : V \times V \to \mathbb{N}$.

b) *Alle Kanten* $(u, v) \in E$ *müssen positive Kapazität* $c(u, v) > 0$ *und alle Nichtkanten* $(u, v) \notin E$ *müssen die Kapazität* $c(u, v) = 0$ *haben.*

Die folgende Abbildung zeigt ein Netzwerk N.

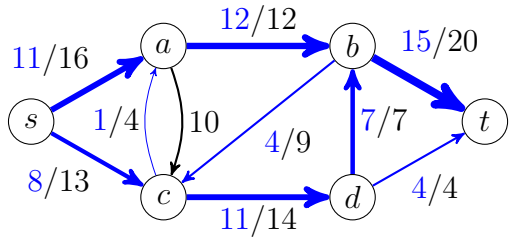


Definition 52. *Ein* ***Fluss*** *für* N *ist eine Funktion* $f : V \times V \to \mathbb{Z}$ *mit*

- $f(u, v) \leq c(u, v)$, *„Kapazitätsbedingung“*
- $f(u, v) = -f(v, u)$, *„Symmetriebedingung“*
- *Für alle* $u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0$. *„Kontinuitätsbedingung“*

Die ***Größe*** *von* f *ist* $|f| = \sum_{v \in V} f(s, v)$.

Die Symmetriebedingung impliziert, dass $f(u, u) = 0$ für alle $u \in V$ ist, d.h. G ist schlingenfrei. Die folgende Abbildung zeigt einen Fluss f für das Netzwerk N.



Wie lässt sich für einen Fluss f in einem Netzwerk N entscheiden, ob er vergrößert werden kann? Diese Frage lässt sich leicht beantworten, falls f der konstante Nullfluss $f = 0$ ist: In diesem Fall genügt es, in $G = (V, E)$ einen Pfad von s nach t zu finden. Andernfalls können wir zu N und f ein Netzwerk N_f konstruieren, so dass jeder Fluss g mit positivem Wert in N_f dazu benutzt werden kann, den Fluss f in N zu vergrößern.

Definition 53. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss für N. Das zugeordnete* ***Restnetzwerk*** *ist $N_f = (V, E_f, s, t, c_f)$, wobei*

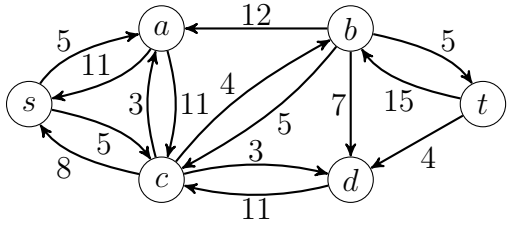
$$c_f(u, v) = c(u, v) - f(u, v)$$

und

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

ist.

Die folgende Abbildung zeigt das Restnetzwerk N_f, das zum Netzwerk N und zum Fluss f gehört.



Definition 54. *Sei $N_f = (V, E_f, s, t, c_f)$ ein Restnetzwerk. Dann heißt jeder s-t-Pfad P in (V, E_f)* Zunahmepfad *für N_f. Die* Kapazität von P in N_f *ist*

$$c_f(P) = \min\{c_f(u, v) \mid (u, v) \text{ liegt auf } P\}$$

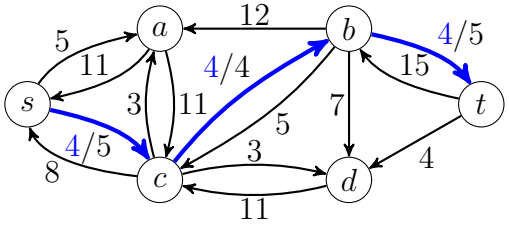
und der zu P gehörige Fluss in N_f *ist*

$$f_P(u, v) = \begin{cases} c_f(P), & (u, v) \text{ liegt auf } P, \\ -c_f(P), & (v, u) \text{ liegt auf } P, \\ 0, & \text{sonst.} \end{cases}$$

$P = (u_0, \ldots, u_k)$ ist also genau dann ein Zunahmepfad in N_f, falls

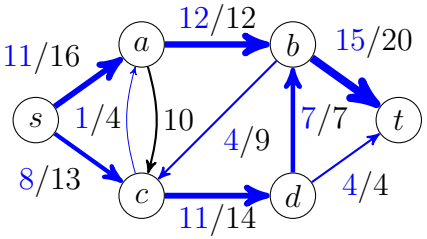
- $u_0 = s$ und $u_k = t$ ist,
- die Knoten $u_0, \ldots, u_k$ paarweise verschieden sind
- und $c_f(u_i, u_{i+1}) > 0$ für $i = 0, \ldots, k-1$ ist.

Die folgende Abbildung zeigt den zum Zunahmepfad $P = s, c, b, t$ gehörigen Fluss f_P in N_f. Die Kapazität von P ist $c_f(P) = 4$.

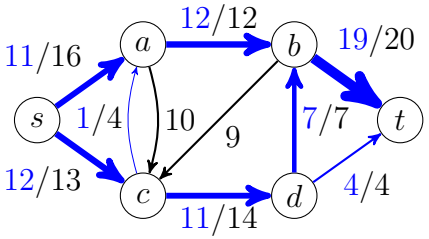


Es ist leicht zu sehen, dass f_P tatsächlich ein Fluss für N_f ist. Durch Addition der beiden Flüsse f und f_P erhalten wir einen größeren Fluss $f' = f + f_P$ für N mit dem Wert $|f'| = |f| + |f_P|$.

Fluss f:



Fluss $f + f_P$:

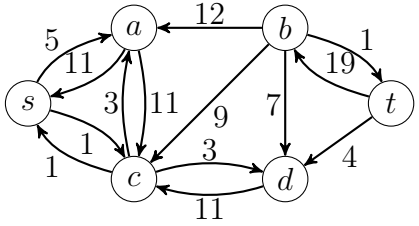


Nun können wir den **Ford-Fulkerson-Algorithmus** angeben.

Algorithmus `Ford-Fulkerson`(V, E, s, t, c)

```
1  for all (u, v) ∈ V × V do
2      f(u, v) := 0
3  while es gibt einen Zunahmepfad P fuer N_f do
4      f := f + f_P
```

Beispiel 55. *Für den neuen Fluss erhalten wir nun folgendes Restnetzwerk:*



In diesem existiert kein Zunahmepfad mehr. ◁

Um zu beweisen, dass der Algorithmus von Ford-Fulkerson tatsächlich einen Maximalfluss berechnet, zeigen wir, dass es nur dann im Restnetzwerk N_f keinen Zunahmepfad mehr gibt, wenn der Fluss f maximal ist. Hierzu benötigen wir den Begriff des Schnitts.

Definition 56. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und seien $S, T \subseteq V$ mit $S \cap T = \emptyset$ und $S \cup T = V$. Dann heißt (S, T) ein* ***Schnitt*** *durch N, falls $s \in S$ und $t \in T$ ist. Die* ***Kapazität*** *eines Schnittes (S, T) ist*

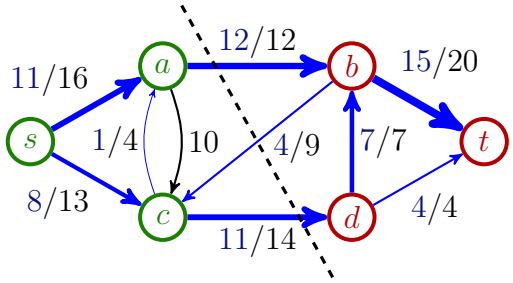
$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$

Ist f ein Fluss für N, so heißt

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v)$$

der ***Fluss durch den Schnitt*** *(S, T).*

Beispiel 57. *Betrachte den Schnitt (S, T) mit $S = \{s, a, c\}$ und $T = \{b, d, t\}$ in folgendem Netzwerk N mit dem Fluss f:*



Dann hat der Schnitt (S, T) die Kapazität

$$c(S,T) = c(a,b) + c(c,d) = 12 + 14 = 26$$

und der Fluss $f(S, T)$ durch den Schnitt (S, T) ist

$$f(S,T) = f(a,b) + f(c,b) + f(c,d) = 12 - 4 + 11 = 19.$$

◁

Lemma 58. *Für jeden Schnitt (S, T) und jeden Fluss f gilt*

$$|f| = f(S,T) \leq c(S,T).$$

Beweis. Die Gleichheit $|f| = f(S, T)$ zeigen wir durch Induktion über $k = \|S\|$.

$k = 1$: In diesem Fall ist $S = \{s\}$ und $T = V - \{s\}$. Daher folgt

$$|f| = \sum_{v \in V - \{s\}} f(s,v) = \sum_{u \in S, v \in T} f(u,v) = f(S,T).$$

$k - 1 \rightsquigarrow k$: Sei (S, T) ein Schnitt mit $\|S\| = k > 1$ und sei $w \in S - \{s\}$. Betrachte den Schnitt (S', T') mit $S' = S - \{w\}$ und $T' = T \cup \{w\}$. Dann gilt

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v) = \sum_{u \in S', v \in T} f(u,v) + \sum_{v \in T} f(w,v)$$

und

$$f(S',T') = \sum_{u\in S', v\in T'} f(u,v) = \sum_{u\in S', v\in T} f(u,v) + \sum_{u\in S'} f(u,w).$$

Wegen $f(w,w) = 0$ ist $\sum_{u\in S'} f(u,w) = \sum_{u\in S} f(u,w)$ und daher

$$\begin{aligned} f(S,T) - f(S',T') &= \sum_{v\in V-S} f(w,v) - \sum_{u\in S} f(u,w) \\ &= \sum_{v\in V} f(w,v) = 0. \end{aligned}$$

Nach Induktionsvoraussetzung folgt somit $f(S,T) = f(S',T') = |f|$.

Schließlich folgt wegen $f(u,v) \le c(u,v)$ die Ungleichung

$$f(S,T) = \sum_{u\in S, v\in T} f(u,v) \le \sum_{u\in S, v\in T} c(u,v) = c(S,T).$$

□

Satz 59 (Min-Cut-Max-Flow-Theorem)**.** *Sei f ein Fluss für ein Netzwerk $N = (V,E,s,t,c)$. Dann sind folgende Aussagen äquivalent:*

1. *f ist maximal.*
2. *In N_f existiert kein Zunahmepfad.*
3. *Es gibt einen Schnitt (S,T) mit $c(S,T) = |f|$.*

Beweis. Die Implikation „1 ⇒ 2“ ist klar, da die Existenz eines Zunahmepfads zu einer Vergrößerung von f führen würde.

Für die Implikation „2 ⇒ 3“ betrachten wir den Schnitt (S,T) mit

$$S = \{u \in V \mid u \text{ ist in } N_f \text{ von } s \text{ aus erreichbar}\} \text{ und } T = V - S.$$

Da in N_f kein Zunahmepfad existiert, gilt dann

- $s \in S$, $t \in T$ und
- $c_f(u,v) = 0$ für alle $u \in S$ und $v \in T$.

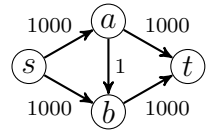
Wegen $c_f(u,v) = c(u,v) - f(u,v)$ folgt somit

$$|f| = f(S,T) = \sum_{u\in S, v\in T} f(u,v) = \sum_{u\in S, v\in T} c(u,v) = c(S,T).$$

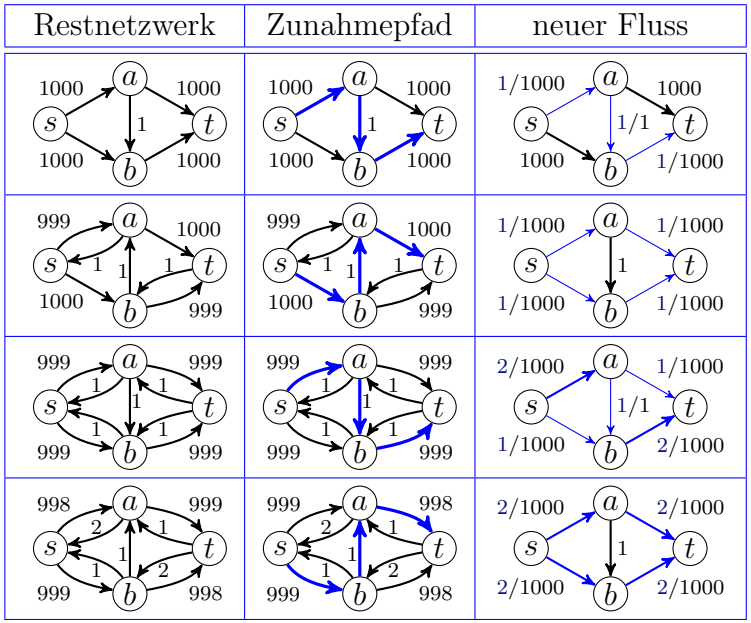
Die Implikation „3 ⇒ 1“ ist wiederum klar, da im Fall $c(S,T) = |f|$ für jeden Fluss f' die Abschätzung $|f'| = f'(S,T) \le c(S,T) = |f|$ gilt. □

Sei $c_0 = \sum_{v\in V} c(s,v)$ die Kapazität des Schnittes (S,T) mit $S = \{s\}$. Dann durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0-mal. Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen.

Die Berechnung des Zunahmepfads P kann durch Breitensuche in Zeit $O(n+m)$ erfolgen. Da sich das Restnetzwerk nur entlang von P ändert, kann es in Zeit $O(n)$ aktualisiert werden. Jeder Durchlauf benötigt also Zeit $O(n+m)$, was auf eine Gesamtlaufzeit von $O(c_0(n+m))$ führt. Da der Wert von c_0 jedoch exponentiell in der Länge der Eingabe (also der Beschreibung des Netzwerkes N) sein kann, ergibt dies keine polynomielle Zeitschranke.



Bei nebenstehendem Netzwerk benötigt Ford-Fulkerson zur Bestimmung des Maximalflusses abhängig von der Wahl der Zunahmepfade zwischen 2 und 2000 Schleifendurchläufe.
Im günstigsten Fall wird zuerst der Zunahmepfad (s,a,t) und dann der Pfad (s,b,t) gewählt. Im ungünstigsten Fall werden abwechselnd die beiden Zunahmepfade (s,a,b,t) und (s,b,a,t) gewählt:



Nicht nur in diesem Beispiel lässt sich die exponentielle Laufzeit wie folgt vermeiden:

- Man betrachtet nur Zunahmepfade mit einer geeignet gewählten Mindestkapazität. Dies führt auf eine Laufzeit, die polynomiell in n, m und $\log c_0$ ist.
- Man bestimmt in jeder Iteration einen kürzesten Zunahmepfad im Restnetzwerk. Diese Vorgehensweise wird als *Edmonds-Karp-Strategie* bezeichnet und führt auf eine Laufzeit von $O(nm^2)$ (unabhängig von der Kapazitätsfunktion).
- Man bestimmt in jeder Iteration einen *blockierenden Fluss* (Algorithmus von Dinic). Dies führt auf eine Laufzeit von $O(n^2m)$. Ein blockierender Fluss im Restnetzwerk ist ein Fluss mit der Eigenschaft, dass es auf jedem kürzesten s-t-Pfad mindestens eine Kante e gibt, die gesättigt ist (d.h. der Fluss durch e schöpft die Kapazität von e voll aus).

3.9 Planare Graphen

Definition 60.

- *Ein Graph G heißt* **planar**, *wenn er so in die Ebene einbettbar ist, dass sich zwei verschiedene Kanten höchstens in ihren Endpunkten berühren.*
- *Dabei werden die Knoten von G als Punkte und die Kanten von G als Verbindungslinien zwischen den zugehörigen Endpunkten dargestellt.*
- *Ein* **ebener** *Graph ist ein in die Ebene eingebetteter Graph.*

Beispiel 61. *Wie die folgenden Einbettungen von K_4 und $K_{2,3}$ in die Ebene zeigen, sind K_4 und $K_{2,3}$ planar.*

◁

Um eine Antwort auf die Frage zu finden, ob auch K_5 und $K_{3,3}$ planar sind, betrachten wir die Gebiete von ebenen Graphen.

Definition 62.

- *Durch die Kanten eines ebenen Graphen wird die Ebene in so genannte* Gebiete *unterteilt.*
- *Nur eines dieser Gebiete ist unbeschränkt und dieses wird als* äußeres Gebiet *bezeichnet.*
- *Die Anzahl der Gebiete von G bezeichnen wir mit $r(G)$ oder kurz mit r.*
- *Der* Rand *eines Gebiets ist die Menge aller Kanten, die an dieses Gebiet grenzen.*

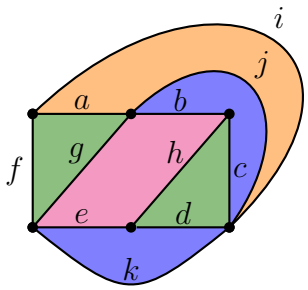
- *Die Anzahl der an ein Gebiet g grenzenden Kanten bezeichnen wir mit $d(g)$, wobei von g eingeschlossene Kanten doppelt gezählt werden.*
- *Die Gesamtzahl aller Inzidenzen von Gebieten und Kanten bezeichnen wir mit $i(G)$. Da hierbei jede Kante mit 2 Inzidenzen zu Buche schlägt, gilt*

$$\sum_g d(g) = i(G) = 2m(G).$$

- *Ein ebener Graph wird durch das Tripel $G = (V, E, R)$ beschrieben, wobei R aus den Rändern aller Gebiete von G besteht.*

Beispiel 63. *Nebenstehender ebener Graph hat 11 Kanten $a, \ldots, k$ und 7 Gebiete mit den Rändern*

$\{a, f, g\}$, $\{a, i, j\}$, $\{b, h, e, g\}$, $\{b, c, j\}$, $\{c, d, h\}$, $\{e, d, k\}$, $\{f, i, k\}$.



◁

Satz 64 (Polyederformel von Euler, 1750)**.** *Sei $G = (V, E, R)$ ein zusammenhängender ebener Graph. Dann gilt*

$$n(G) - m(G) + r(G) = 2. \qquad (*)$$

Beweis. Wir führen den Beweis durch Induktion über die Kantenzahl $m(G) = m$.

$m = 0$**:** Da G zusammenhängend ist, muss dann $n = 1$ sein.

Somit ist auch $r = 1$, also $(*)$ erfüllt.

$m - 1 \rightsquigarrow m$**:** Sei G ein zusammenhängender ebener Graph mit m Kanten.

Falls G ein Baum ist, können wir ein Blatt entfernen. Dies führt auf einen Graphen mit $n - 1$ Knoten, $m - 1$ Kanten und r Gebieten. Nach IV gilt daher $(n - 1) - (m - 1) + r = 2$, so dass $(*)$ erfüllt ist.

Ist G kein Baum, so können wir eine Kante auf einem Kreis entfernen. Der resultierende Graph hat n Knoten, $m-1$ Kanten und $r-1$ Gebiete. Nach IV gilt daher $n-(m-1)+(r-1) = 2$, so dass $(*)$ auch in diesem Fall erfüllt ist. □

Korollar 65. *Sei $G = (V, E)$ ein planarer Graph mit $n \geq 3$ Knoten. Dann ist $m \leq 3n - 6$.*

Beweis. O.B.d.A. sei G zusammenhängend. Wir betrachten eine beliebige planare Einbettung von G. Da $n \geq 3$ ist, ist jedes Gebiet g von $d(g) \geq 3$ Kanten umgeben. Daher ist $2m = i = \sum_g d(g) \geq 3r$ bzw. $r \leq 2m/3$. Eulers Formel liefert

$$m = n + r - 2 \leq n + 2m/3 - 2 \text{ bzw. } (1 - 2/3)m \leq n - 2,$$

was $m \leq 3n - 6$ impliziert. □

Korollar 66. *K_5 ist nicht planar.*

Beweis. Wegen $n = 5$, also $3n - 6 = 9$, und wegen $m = \binom{5}{2} = 10$ gilt $m \not\leq 3n - 6$. □

Korollar 67. *Sei $G = (V, E)$ ein dreiecksfreier, planarer Graph mit $n \geq 3$ Knoten. Dann ist $m \leq 2n - 4$.*

Beweis. Wir betrachten eine beliebige planare Einbettung von G, wobei wir wieder o.B.d.A. annehmen, dass G zusammenhängend ist. Da G dreiecksfrei ist, ist jedes Gebiet von $d(g) \geq 4$ Kanten umgeben. Daher ist $2m = i = \sum_g d(g) \geq 4r$ bzw. $r \leq m/2$. Eulers Formel liefert $m = n + r - 2 \leq n + m/2 - 2$ bzw. $m/2 \leq n - 2$, was $m \leq 2n - 4$ impliziert. □

Korollar 68. *$K_{3,3}$ ist nicht planar.*

Beweis. Wegen $n = 6$, also $2n - 4 = 8$, und wegen $m = 3 \cdot 3 = 9$ gilt $m \not\leq 2n - 4$. □

Eine weitere interessante Folgerung ist die Existenz eines Knotens v vom Grad $deg(v) \leq 5$ in einem planaren Graphen.

Lemma 69. *Sei G ein planarer Graph. Dann ist der Minimalgrad $\delta(G) \leq 5$.*

Beweis. Für $n \leq 6$ ist die Behauptung klar. Für $n > 6$ impliziert die Annahme $\delta(G) \geq 6$ die Ungleichung

$$m = \tfrac{1}{2} \sum_{u \in V} deg(u) \geq \tfrac{1}{2} \sum_{u \in V} 6 = 3n.$$

Dies widerspricht aber der Ungleichung $m \leq 3n - 6$. □

Beispiel 70. *Es gilt $\chi(E_n) = 1$, $\chi(K_{n,m}) = 2$, $\chi(K_n) = n$ und*

$$\chi(C_n) = \begin{cases} 2, & n \text{ gerade,} \\ 3, & \text{sonst.} \end{cases}$$

Bereits im 19. Jahrhundert wurde die Frage aufgeworfen, wie viele Farben höchstens benötigt werden, um eine Landkarte so zu färben, dass aneinander grenzende Länder unterschiedliche Farben erhalten. Offensichtlich lässt sich eine Landkarte in einen planaren Graphen transformieren, indem man für jedes Land einen Knoten zeichnet und benachbarte Länder durch eine Kante verbindet. Länder, die sich nur in einem Punkt berühren, gelten nicht als benachbart.

Die Vermutung, dass 4 Farben ausreichen, wurde 1878 von Kempe „bewiesen“ und erst 1890 entdeckte Heawood einen Fehler in Kempes „Beweis“. Übrig blieb der *5-Farben-Satz*. Der *4-Farben-Satz* wurde erst 1976 von Appel und Haken bewiesen. Hierbei handelt es sich jedoch nicht um einen Beweis im klassischen Sinne, da zur Überprüfung der vielen auftretenden Spezialfälle Computer eingesetzt wurden.

Satz 71 (Appel, Haken 1976)**.** *Jeder planare Graph ist 4-färbbar.*

Der Beweis des 4-Farben-Satzes von Appel und Haken liefert einen effizienten 4-Färbungsalgorithmus für planare Graphen mit einer Laufzeit von $O(n^4)$.

In 1997 fanden Robertson, Sanders, Seymour und Thomas eine Möglichkeit, den Beweis von Appel und Haken zu vereinfachen. Dieser Beweis führt zwar zu einem deutlich schnelleren Algorithmus zur 4-Färbung von planaren Graphen mit quadratischer Laufzeit $O(n^2)$. Aber auch er lässt sich nicht ohne Computer-Unterstützung verifizieren.

Satz 72 (Kempe 1878, Heawood 1890)**.** *Jeder planare Graph ist 5-färbbar.*

Beweis. Wir führen den Beweis durch Induktion über die Knotenzahl n von G.

$n = 1$**:** Klar.

$n - 1 \rightsquigarrow n$**:** Da G planar ist, existiert ein Knoten v mit $deg(v) \leq 5$. Zunächst entfernen wir v aus G. Falls v fünf Nachbarn hat, sind diese nicht alle paarweise adjazent, und wir verschmelzen zusätzlich zwei nicht durch eine Kante verbundene Nachbarn von v zu einem Knoten.

Da der resultierende Graph G' maximal $n - 1$ Knoten hat und wie G planar ist, hat G' nach IV eine 5-Färbung c. Trennen wir nun die verschmolzenen Nachbarn wieder, so ist c immer noch eine 5-Färbung, die die Nachbarn von v mit höchstens 4 Farben färbt. Daher können wir v mit der verbliebenen Farbe färben.

□