

Einleitung



Wer Bildhauer¹ werden will, muss eine Reihe von Grundtechniken erlernen: woher man die geeigneten Steine bezieht, wie man sie bewegt, wie man mit dem Meißel arbeitet, wie man ein Gerüst aufbaut, Wer die Grundtechniken beherrscht, ist noch lange kein berühmter Künstler, aber selbst jemand mit außergewöhnlichem Talent wird kaum ein erfolgreicher Künstler werden, wenn er die Grundtechniken nicht kennt. Natürlich muss er nicht alle beherrschen, bevor er die erste Skulptur gestaltet. Aber er muss stets bereit sein, zu den Grundtechniken zurückzukehren, um sie immer besser beherrschen zu lernen.

Dieses einleitende Kapitel spielt für dieses Buch eine ähnliche Rolle. Wir stellen hier grundlegende Begriffe und Methoden vor, mit deren Hilfe wir in späteren Kapiteln Algorithmen leichter besprechen und analysieren können. Man muss dieses Kapitel nicht von A bis Z durcharbeiten, bevor man die nachfolgenden Kapitel liest. Wir empfehlen dem Leser, beim ersten Durchgang das Material bis einschließlich Abschnitt 2.3 genau zu studieren und die restlichen Abschnitte einmal durchzusehen. Abschnitt 2.1 macht den Anfang mit Notation und Terminologie, die uns helfen wird, in knapper Form über die Komplexität von Algorithmen zu reden. In Abschnitt 2.2 wird ein einfaches Maschinenmodell vorgestellt, das es ermöglicht, durch eine abstrakte Sichtweise die vielfältigen Komplikationen zu vermeiden, die sich bei der Berücksichtigung der Eigenschaften echter Hardware ergeben würden. Das Modell ist konkret genug, um nützliche Vorhersagen zu liefern, und abstrakt genug, um damit elegante Überlegungen anzustellen. In Abschnitt 2.3 wird eine Pseudocode-Notation eingeführt, die einer höheren Programmiersprache ähnelt und für das Aufschreiben von Algorithmen viel bequemer zu handhaben ist als die Maschinensprache unseres abstrakten Modells. Die Benutzung von Pseudocode ist außerdem bequemer als die einer echten Programmiersprache, da wir dann abstrakte Begriffe aus der Mathematik verwenden können, ohne uns den Kopf darüber zu zerbrechen, wie man

¹ Die obige Abbildung des Steinrings von Stonehenge ist [170] entnommen.

sie übersetzen kann, damit sie auf echten Maschinen benutzt werden können. Wir werden Programme häufig mit Anmerkungen versehen, um sie besser lesbar zu machen und um das Führen von Korrektheitsbeweisen zu erleichtern. Techniken für solche Beweise sind der Gegenstand von Abschnitt 2.4. In Abschnitt 2.5 wird ein erstes Beispiel umfassend besprochen: Binäre Suche in einem geordneten Array. In Abschnitt 2.6 werden mathematische Techniken für die Komplexitätsanalyse von Programmen besprochen, insbesondere für die Analyse von geschachtelten Schleifen und rekursiven Prozeduraufrufen. Für die Analyse von Algorithmen im mittleren Fall werden weitere Techniken benötigt; mit diesen befasst sich Abschnitt 2.7. Randomisierte Algorithmen, vorgestellt in Abschnitt 2.8, führen während des Ablaufs Zufallsexperimente („Münzwürfe“) durch. Abschnitt 2.9 behandelt Graphen, einen Begriff, der im Rest des Buches eine große Rolle spielen wird. In Abschnitt 2.10 wird die Frage diskutiert, wann man einen Algorithmus effizient nennen soll, und die Komplexitätsklassen **P** und **NP** sowie die wichtige Klasse der **NP**-vollständigen Probleme werden vorgestellt. Das Kapitel schließt, wie jedes Kapitel in diesem Buch, mit einem Abschnitt zu Implementierungsaspekten (Abschnitt 2.11) und einem mit historischen Anmerkungen und weiteren Ergebnissen (Abschnitt 2.12).

2.1 Asymptotische Notation

Die Analyse eines Algorithmus dient hauptsächlich dazu, zuverlässige Aussagen über sein Verhalten zu gewinnen, z. B. Schranken für die Rechenzeit, die zugleich genau, knapp, allgemein und leicht verständlich sein sollen. Natürlich ist es schwierig, all diesen Anforderungen gleichzeitig gerecht zu werden. Um beispielsweise die Rechenzeit eines Algorithmus zu erfassen, kann man T als eine Funktion auffassen, die die Menge \mathcal{I} aller möglichen Eingaben (oder *Instanzen*) in die Menge $\mathbb{R}_{>0}$ der positiven reellen Zahlen abbildet. Für jede Instanz I des Problems ist dann $\text{time}(I)$ die Rechenzeit auf I . Eine solche Detailgenauigkeit führt aber zu einer so überwältigenden Menge an Information, dass man darüber niemals eine brauchbare Theorie entwickeln könnte. Eine nützliche Theorie muss das (Rechenzeit-)Verhalten eines Algorithmus von einem allgemeineren Standpunkt aus betrachten.

Wir unterteilen die Menge aller Eingaben in Klassen „ähnlicher“ Eingaben und fassen das (Rechenzeit-)Verhalten des Algorithmus auf Eingaben einer Klasse in einer einzigen Zahl zusammen. Das nützlichste Kriterium für eine Klasseneinteilung ist die *Größe* einer Eingabe. Normalerweise kann man jeder Eingabe auf natürliche Art eine Größe zuordnen. Die Größe einer ganzen Zahl ist die Anzahl der Ziffern ihrer Binärdarstellung; die Größe einer Menge ist die Anzahl ihrer Elemente. Die Größe einer Eingabe ist immer eine natürliche Zahl. Manchmal verwendet man mehr als einen Parameter, um die Größe einer Eingabe anzugeben; beispielsweise ist es üblich, die Größe eines Graphen durch die Anzahl seiner Knoten und seiner Kanten zu charakterisieren. Diese kleine Komplikation werden wir hier zunächst außer Acht lassen. Die Größe der Eingabe I wird mit $\text{size}(I)$ bezeichnet, und mit \mathcal{I}_n die Menge aller Eingaben der Größe n , für $n \in \mathbb{N}$. Für die Eingaben der Größe n fragen wir nach

maximalen, minimalen und mittleren Ausführungszeiten:²

$$\begin{aligned} \textbf{schlechtester Fall (worst case): } T(n) &= \max \{ \text{time}(I) : I \in \mathcal{I}_n \} \\ \textbf{bester Fall (best case): } T(n) &= \min \{ \text{time}(I) : I \in \mathcal{I}_n \} \\ \textbf{mittlerer Fall (average case): } T(n) &= \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} \text{time}(I) . \end{aligned}$$

Die interessanteste Größe ist dabei die Ausführungszeit im schlechtesten Fall, weil sie die umfassendste Garantie für das Verhalten des Algorithmus darstellt. Ein Vergleich der Rechenzeiten im besten und im schlechtesten Fall sagt uns, wie stark die Ausführungszeit für Eingaben aus einer Größenklasse variieren kann. Wenn die Abweichung zwischen bestem und schlechtestem Fall sehr groß ist, liefert eventuell eine Analyse der Rechenzeit im mittleren Fall genauere Einsichten in das wirkliche Rechenzeitverhalten des Algorithmus. Ein Beispiel hierfür werden wir in Abschnitt 2.7 sehen.

Wir gehen in der Vergrößerung und damit Reduktion der Daten noch einen Schritt weiter: Wir konzentrieren uns auf die Untersuchung der *Wachstumsordnung*, oder die *asymptotische Analyse*, der Rechenzeit. Zwei Funktionen $f(n)$ und $g(n)$ haben die *gleiche Wachstumsordnung*, wenn es positive Konstanten c und d gibt, so dass für genügend große n die Ungleichung $c \leq f(n)/g(n) \leq d$ gilt. Die Funktion $f(n)$ *wächst schneller* als $g(n)$, wenn für jede positive Konstante c gilt, dass für alle genügend großen n die Ungleichung $f(n) \geq c \cdot g(n)$ erfüllt ist. Beispielsweise haben die Funktionen n^2 , $n^2 + 7n$, $5n^2 - 7n$ und $n^2/10 + 10^6n$ alle die gleiche Wachstumsordnung. Weiter wachsen diese Funktionen alle schneller als die Funktion $n^{3/2}$, die wiederum schneller als $n \log n$ wächst. Die Wachstumsordnung bezieht sich auf das Verhalten für große n . Ebenso soll das Wort „asymptotisch“ in „asymptotische Analyse“ hervorheben, dass es um das Verhalten für große n geht.

Weshalb interessieren wir uns nur für Wachstumsordnungen und das Verhalten für große Eingabegrößen n ? Nun, man entwirft effiziente Algorithmen meistens zu dem Zweck, große Instanzen bearbeiten und lösen zu können. Wenn die Rechenzeit eines Algorithmus A eine kleinere Wachstumsordnung hat als die eines anderen Algorithmus B , wird für große n Algorithmus A überlegen sein. Auch die Tatsache, dass unser Maschinenmodell eine Abstraktion ist und reale Rechenzeiten nur bis auf einen (maschinenabhängigen) konstanten Faktor vorhersagen kann, legt es nahe, keinen Unterschied zwischen Algorithmen zu machen, deren Rechenzeiten dieselbe Wachstumsordnung haben. Dass wir uns auf die Wachstumsordnungen zurückziehen, hat den erfreulichen Nebeneffekt, dass wir die Rechenzeiten von Algorithmen durch sehr einfache Funktionen charakterisieren können. In den Abschnitten, die sich mit Implementierungsaspekten befassen, werden wir jedoch regelmäßig genauer hinschauen und dann den Bereich der asymptotischen Analyse verlassen. Auch sollte sich der Leser bei der Verwendung eines Algorithmus aus diesem Buch immer fragen, ob die asymptotische Sichtweise gerechtfertigt ist.

² Wir werden immer sicherstellen, dass die Menge $\{\text{time}(I) : I \in \mathcal{I}_n\}$ ein Minimum und ein Maximum hat und dass \mathcal{I}_n endlich ist, wenn es um Mittelwerte gehen soll.

Die folgenden Definitionen gestatten es uns, präzise Überlegungen über das *asymptotische Verhalten* von Funktionen anzustellen. Mit $f(n)$ und $g(n)$ seien dabei Funktionen bezeichnet, die natürliche Zahlen auf nichtnegative reelle Zahlen abbilden. Wir definieren:

$$\begin{aligned} O(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}, \\ \Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}, \\ \Theta(f(n)) &= O(f(n)) \cap \Omega(f(n)), \\ o(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}, \\ \omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}. \end{aligned}$$

Die linken Seiten liest man als „groß-O von $f(n)$ “, „groß-Omega von $f(n)$ “, „Theta von $f(n)$ “, „klein-o von $f(n)$ “ bzw. „klein-omega von $f(n)$ “. Man beachte, dass mit „ $f(n)$ “ in „ $O(f(n))$ “ und „ $g(n)$ “ in „ $\{g(n) : \dots\}$ “ die Funktionen $f(n)$ bzw. $g(n)$ bezeichnet werden – es wird nur deutlich gemacht, dass diese Funktionen von der Variablen n abhängen. In Bedingungen wie „ $\forall n \geq n_0 : g(n) \leq c \cdot f(n)$ “ sind hingegen die Funktionswerte für ein n gemeint.

Wir betrachten einige Beispiele. Die Menge $O(n^2)$ enthält die Funktionen, die höchstens quadratisch wachsen, die Menge $o(n^2)$ die Funktionen, die langsamer als quadratisch wachsen, und $o(1)$ enthält die Funktionen, die für wachsendes n gegen Null streben. Dabei steht „1“ für die konstante Funktion $n \mapsto 1$, die überall den Wert 1 hat. Damit liegt eine Funktion $f(n)$ in $o(1)$, wenn $f(n) \leq c \cdot 1$ für jedes positive c gilt, wenn nur n genügend groß ist, d. h., wenn $f(n)$ für wachsendes n gegen Null strebt. Allgemein ist $O(f(n))$ die Menge all der Funktionen, die „nicht schneller wachsen als“ $f(n)$; ähnlich ist $\Omega(f(n))$ die Menge der Funktionen, die „mindestens so schnell wachsen wie“ $f(n)$. Zum Beispiel liegt beim Algorithmus von Karatsuba für die Multiplikation ganzer Zahlen die asymptotische Rechenzeit im schlechtesten Fall in $O(n^{1.585})$, wohingegen die asymptotische Rechenzeit der Schulmethode in $\Omega(n^2)$ liegt. Daher können wir sagen, dass der Algorithmus von Karatsuba asymptotisch schneller ist als die Schulmethode. Die Notation $o(f(n))$ („klein-o“ von $f(n)$) bezeichnet die Menge aller Funktionen, die „strikt langsamer wachsen als“ $f(n)$. Ihr Gegenstück, die Notation $\omega(f(n))$ („klein-omega von $f(n)$ “), wird eher selten benutzt und ist hier nur der Vollständigkeit halber aufgeführt.

Die meisten Algorithmen in diesem Buch haben Rechenzeitschranken, die sich als ein Polynom oder als eine logarithmische Funktion oder als Produkt solcher Funktionen schreiben lassen. Am Beispiel der Polynome machen wir nun den Leser mit einigen grundlegenden Umformungen für unsere asymptotische Notation vertraut.

Lemma 2.1. Sei $p(n) = \sum_{i=0}^k a_i n^i$ ein Polynom mit reellen Koeffizienten, wobei $a_k > 0$ gilt. Dann ist $p(n) \in \Theta(n^k)$.

Beweis. Es genügt, die beiden Beziehungen $p(n) \in O(n^k)$ und $p(n) \in \Omega(n^k)$ zu beweisen. Wir beobachten zunächst, dass für $n > 0$

$$p(n) \leq \sum_{i=0}^k |a_i| n^i \leq n^k \sum_{i=0}^k |a_i|$$

gilt; daraus folgt $p(n) \leq (\sum_{i=0}^k |a_i|) n^k$ für alle positiven n . Daher gilt $p(n) \in O(n^k)$.

Nun setzen wir $A = \sum_{i=0}^{k-1} |a_i|$. Für alle $n > 0$ gilt

$$p(n) \geq a_k n^k - A n^{k-1} = \frac{a_k}{2} n^k + n^{k-1} \left(\frac{a_k}{2} n - A \right)$$

und daher $p(n) \geq (a_k/2) n^k$ für $n > 2A/a_k$. Wir wählen $c = a_k/2$ und $n_0 = 2A/a_k$, und erhalten mit der Definition von $\Omega(n^k)$, dass $p(n) \in \Omega(n^k)$ gilt. \square

Aufgabe 2.1. Richtig oder falsch? (a) $n^2 + 10^6 n \in O(n^2)$; (b) $n \log n \in O(n)$; (c) $n \log n \in \Omega(n)$; (d) $\log n \in o(n)$.

Die asymptotische Notation wird bei der Algorithmenanalyse häufig benutzt, und es ist bequem, die präzise Definition etwas flexibler zu handhaben, indem man Bezeichnungen für Mengen von Funktionen (wie $O(n^2)$) einfach so benutzt, als ob es sich um gewöhnliche Funktionen handeln würde. Insbesondere schreiben wir immer $h(n) = O(f(n))$ anstelle von $h(n) \in O(f(n))$ und $O(h(n)) = O(f(n))$ anstelle von $O(h(n)) \subseteq O(f(n))$, z. B.:

$$3n^2 + 7n = O(n^2) = O(n^3) .$$

Folgen von „Gleichungen“ mit der O-Notation muss man also als Elementbeziehungen und Inklusionen auffassen und als solche immer nur von links nach rechts lesen.

Für eine Funktion $h(n)$, Mengen F und G von Funktionen, und einen Operator \diamond (wie $+$, \cdot oder $/$) soll $F \diamond G$ eine Abkürzung für $\{f(n) \diamond g(n) : f(n) \in F, g(n) \in G\}$ sein, und $h(n) \diamond F$ soll für $\{h(n)\} \diamond F$ stehen. Mit dieser Vereinbarung bezeichnet also $f(n) + o(f(n))$ die Menge aller Funktionen $f(n) + g(n)$ mit der Eigenschaft, dass $g(n)$ strikt langsamer wächst als $f(n)$, d. h., dass der Quotient $(f(n) + g(n))/f(n)$ für $n \rightarrow \infty$ gegen 1 strebt. Äquivalent kann man auch $(1 + o(1))f(n)$ schreiben. Diese Notation benutzen wir, wenn wir die Rolle von $f(n)$ als „führendem Term“ betonen wollen, gegenüber dem „Terme niedrigerer Ordnung“ ignoriert werden können.

Lemma 2.2. Für die O-Notation gelten die folgenden Regeln:

$$\begin{aligned} cf(n) &= \Theta(f(n)), \text{ für jede positive Konstante } c, \\ f(n) + g(n) &= \Omega(f(n)), \\ f(n) + g(n) &= O(f(n)), \text{ wenn } g(n) = O(f(n)), \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)). \end{aligned}$$

Aufgabe 2.2. Beweisen Sie Lemma 2.2.

Aufgabe 2.3. Verschärfen Sie Lemma 2.1, indem Sie zeigen, dass sogar $p(n) = a_k n^k + o(n^k)$ gilt.

Aufgabe 2.4. Beweisen Sie, dass $n^k = o(c^n)$ gilt, für ganzzahlige k und beliebige $c > 1$. In welcher Relation steht $n^{\log \log n}$ zu n^k und c^n ?

2.2 Das Maschinenmodell

Im Jahr 1945 schlug John von Neumann (Abb. 2.1) eine einfache, aber mächtige Rechnerarchitektur vor [215]. Die eingeschränkten Hardwaremöglichkeiten der damaligen Zeit brachten ihn zu einem eleganten Entwurf, der sich auf das Wesentliche beschränkte; andernfalls wäre eine Realisierung nicht möglich gewesen. In den Jahren seit 1945 hat sich zwar die Hardwaretechnologie dramatisch weiterentwickelt, aber das Programmiermodell, das sich aus von Neumanns Entwurf ergab, ist so elegant und mächtig, dass es heute noch die Grundlage für den Großteil der modernen Programmierung darstellt. Im Normalfall funktionieren Programme, die für das von-Neumann-Modell geschrieben werden, auch auf der viel komplexeren Hardware der heutigen Computer recht gut.



Abb. 2.1. John von Neumann, * 28.12.1903 in Budapest, † 8.2.1957 in Washington, DC.

Für die Analyse von Algorithmen benutzt man eine Variante des von-Neumann-Rechners, die *RAM* („random access machine“: Maschine mit wahlfreiem Speicherzugriff) oder *Registermaschine* genannt wird. Dieses Modell wurde 1963 von Shepherdson und Sturgis vorgeschlagen [195]. Dabei handelt es sich um einen *sequentiellen* Rechner mit uniformem Speicher, d. h., es gibt nur eine Recheneinheit (CPU), und jeder Speicherzugriff kostet genau gleich viel Zeit. Der Speicher („memory“ oder „store“) besteht aus unendlich vielen Speicherzellen $S[0], S[1], S[2], \dots$; zu jedem Zeitpunkt ist nur eine endliche Anzahl davon in Gebrauch. Zusätzlich zu diesem Hauptspeicher besitzt eine RAM eine kleine, konstante Anzahl von *Registern* R_1, \dots, R_k .

In den Zellen und den Registern werden „kleine“ ganze Zahlen gespeichert, die auch (*Maschinen*-) *Wörter* genannt werden. In unseren Überlegungen zur ganzzahligen Arithmetik in Kap. 1 hatten wir angenommen, dass „klein“ das gleiche wie „einziffrig“ bedeutet. Es ist jedoch vernünftiger und auch bequemer anzunehmen, dass es von der Eingabegröße abhängt, was „klein“ bedeutet. Unsere Standardannahme wird sein, dass eine ganze Zahl in einer Zelle gespeichert sein kann, solange ihre Größe durch ein Polynom in der Größe der Eingabe beschränkt ist. Die Binärdarstellung solcher Zahlen benötigt eine Anzahl von Bits, die logarithmisch in der Größe der Eingabe ist. Diese Annahme ist vernünftig, weil wir den Inhalt einer Speicherzelle immer auf logarithmisch viele Zellen aufteilen könnten, von denen jede ein Bit speichern kann, ohne dass Zeit- und Speicherbedarf um mehr als einen logarithmischen Faktor zunehmen. Da Register für die Adressierung von Speicherzellen benutzt werden, wird man verlangen, dass jede in einer Berechnung vorkommenden Adresse in einem Register Platz findet. Hierfür ist eine polynomielle Größenbeschränkung ausreichend.

Die Annahme einer logarithmischen Beschränkung der Bitlänge der gespeicherten Zahlen ist aber auch notwendig: Wenn man zuliebe, dass in einer Zelle Zahlen

beliebiger Größe gespeichert werden können, würden sich in manchen Fällen Algorithmen mit absurd kleinem Zeitbedarf ergeben. Beispielsweise könnte man mit n nacheinander ausgeführten Quadrierungen aus der Zahl 2 (zwei Bits) eine mit 2^n Bits erzeugen. Man beginnt mit $2 = 2^1$, quadriert einmal, um $2^2 = 4$ zu erhalten, nochmaliges Quadrieren liefert $16 = 2^{2 \cdot 2}$ usw. Nach n -maligem Quadrieren ist die Zahl 2^{2^n} entstanden.

Unser Modell lässt eine bestimmte, eingeschränkte Art von Parallelverarbeitung zu: einfache Operationen auf einer logarithmischen Anzahl von Bits können in konstanter Zeit ausgeführt werden.

Eine RAM kann ein (Maschinen-)Programm ausführen. Ein solches *Programm* ist dabei eine Liste von Maschinenbefehlen, durchnummeriert von 1 bis zu einer Zahl ℓ . Die Einträge der Liste heißen *Zeilen* des Programms. Das Programm steht in einem Programmspeicher. Unsere RAM kann folgende Maschinenbefehle ausführen:

- $R_i := S[R_j]$ *lädt* den Inhalt der Speicherzelle, deren Index in Register R_j steht, in Register R_i .
- $S[R_j] := R_i$ *schreibt* den Inhalt von Register R_i in die Speicherzelle, deren Index in Register R_j steht.
- $R_i := R_j \odot R_h$ führt auf den Inhalten von Registern R_j und R_h eine binäre Operation \odot aus und speichert das Ergebnis in Register R_i . Dabei gibt es für „ \odot “ eine Reihe von Möglichkeiten. Die *arithmetischen* Operationen sind wie üblich $+$, $-$ und $*$; sie interpretieren die Registerinhalte als ganze Zahlen. Die Operationen **div** und **mod**, ebenfalls für ganze Zahlen, liefern den Quotienten bzw. den Rest bei der ganzzahligen Division. Die *Vergleichsoperationen* \leq , $<$, $>$ und \geq für ganze Zahlen liefern als Ergebnis einen Wahrheitswert, also *true* ($= 1$) oder *false* ($= 0$). Daneben gibt es auch die bitweise auszuführenden Operationen $|$ (logisches Oder, OR), $\&$ (logisches Und, AND) und \oplus (exklusives Oder, XOR); sie interpretieren Registerinhalte als Bitstrings. Die Operationen \gg (Rechtsshift) und \ll (Linksshift) interpretieren das erste Argument als Bitstring und das zweite als nichtnegativen Verschiebewert. Die *logischen* Operationen \wedge und \vee verarbeiten die *Wahrheitswerte* 1 und 0. Wir können auch annehmen, dass es Operationen gibt, die die in einem Register gespeicherten Bits als Gleitkommazahl interpretieren, d. h. als endlichen Näherungswert für eine reelle Zahl.
- $R_i := \odot R_j$ führt auf Register R_j eine *unäre* Operation \odot aus und speichert das Ergebnis in Register R_i . Dabei sind die Operationen $-$ (für ganze Zahlen), \neg (logische Negation für Wahrheitswerte, NOT) und \sim (bitweise Negation für Bitstrings) vorgesehen.
- $R_i := C$ weist dem Register R_i einen *konstanten* Wert C zu.
- $\text{JZ } k, R_i$ setzt die Berechnung in Programmzeile k fort, wenn Register R_i den Inhalt 0 hat (*Verzweigung* oder *bedingter Sprung*), andernfalls in der nächsten Programmzeile.³
- $\text{J } k$ setzt die Berechnung in Programmzeile k fort (*unbedingter Sprung*).

³ Das Sprungziel k kann, falls nötig, auch als Inhalt $S[R_j]$ einer Speicherzelle angegeben sein.

Ein solches Programm wird auf einer gegebenen Eingabe Schritt für Schritt ausgeführt. Die Eingabe steht zu Beginn in den Speicherzellen $S[1]$ bis $S[R_1]$; die erste ausgeführte Programmzeile ist Zeile 1. Mit Ausnahme der Sprungbefehle JZ und J folgt auf die Ausführung einer Programmzeile stets die Ausführung der darauffolgenden Programmzeile. Die Ausführung des Programms endet, wenn eine Zeile ausgeführt werden soll, deren Nummer außerhalb des Bereichs $1..l$ liegt.

Wir legen den Zeitaufwand für die Ausführung eines Programms auf einer Eingabe auf denkbar einfache Art fest: *Die Ausführung eines Maschinenbefehls dauert genau eine Zeiteinheit*. Die Gesamtausführungszeit oder *Rechenzeit* eines Programms ist dann einfach die Gesamtzahl aller ausgeführten Befehle.

Es ist wichtig, sich klarzumachen, dass die RAM eine Abstraktion ist; man darf das Modell nicht mit wirklichen Rechnern verwechseln. Insbesondere haben echte Rechner einen endlichen Speicher und eine feste Anzahl von Bits pro Register bzw. Speicherzelle (z. B. 32 oder 64). Im Gegensatz hierzu wachsen („skalieren“) Wortlänge und Speichergröße einer RAM mit der Eingabegröße. Wenn man so will, kann man dies als Abstraktion der historischen Entwicklung der Rechner ansehen: Mikroprozessoren hatten nacheinander Wortlängen von 4, 8, 16, 32 und 64 Bits. Mit Wörtern der Länge 64 kann man einen Speicher der Größe 2^{64} adressieren. Daher ist, auf der Basis heutiger Preise für Speichermedien, die Speichergröße durch die Kosten und nicht durch die Länge der Adressen begrenzt. Interessanterweise traf diese Aussage auch zu, als 32-Bit-Wörter eingeführt wurden!

Auch unser Modell für die Komplexität von Berechnungen stellt insofern eine grobe Vereinfachung dar, als moderne Prozessoren versuchen, viele Instruktionen gleichzeitig auszuführen. Zu welcher Zeitersparnis dies führt, hängt von Faktoren wie Datenabhängigkeiten zwischen aufeinanderfolgenden Operationen ab. Infolgedessen ist es gar nicht möglich, einer Operation feste Kosten zuzuordnen. Dieser Effekt tritt besonders bei Speicherzugriffen zutage. Die im schlechtesten Fall für einen Zugriff auf den Hauptspeicher benötigte Zeit kann mehrere Hundert mal größer sein als die im besten Fall! Dies liegt daran, dass moderne Prozessoren versuchen, häufig benutzte Daten in *Cachespeichern* zu halten, das sind vergleichsweise kleine, schnelle Speichereinheiten, die eng an die Prozessoren angebunden sind. Welche Rechenzeitersparnis durch Cachespeicher erzielt wird, hängt sehr stark von der Architektur, vom Programm und sogar von der konkreten Eingabe ab.

Wir könnten versuchen, ein sehr genaues Kostenmodell zu entwickeln, aber dies ginge am Ziel vorbei. Es ergäbe sich ein sehr komplexes und schwer handhabbares Modell. Selbst eine erfolgreiche Analyse würde zu einer monströsen Formel führen, die von vielen Parametern abhängt, die sich dann auch noch mit jeder neuen Prozessorgeneration ändern. Obgleich die in einer solchen Formel enthaltene Information sehr präzise wäre, wäre sie allein aufgrund ihrer Komplexität unbrauchbar. Daher gehen wir gleich zum anderen Extrem, eliminieren sämtliche Modellparameter und nehmen einfach an, dass die Ausführung eines Maschinenbefehls genau eine Zeiteinheit beansprucht. Das führt dazu, dass konstante Faktoren in unserem Modell eigentlich keine Rolle spielen – ein weiterer Grund dafür, dass wir meistens bei der asymptotischen Analyse von Algorithmen bleiben. Zum Ausgleich gibt es in jedem Kapitel einen Abschnitt zu Implementierungsaspekten, in denen Implemen-

tierungsvarianten und „*trade-offs*“ diskutiert werden, also Abhängigkeiten zwischen dem Bedarf an verschiedenen Ressourcen wie Zeit und Speicherplatz.

2.2.1 Hintergrundspeicher

Eine RAM und ein realer Rechner unterscheiden sich am dramatischsten in der Speicherstruktur: einem uniformen Speicher in der RAM steht eine komplexe Speicherhierarchie im realen Rechner gegenüber. In den Abschnitten 5.7, 6.3 und 7.6 werden wir Algorithmen betrachten, die speziell auf große Datenmengen zugeschnitten sind, die in langsamen (Hintergrund-)Speichern wie etwa Platten gehalten werden müssen. Zur Untersuchung dieser Algorithmen benutzen wir das *Externspeichermodell*.

Das Externspeichermodell ähnelt dem RAM-Modell, unterscheidet sich von diesem aber in der Speicherstruktur. Der (schnelle) interne Speicher (Hauptspeicher) besteht nur aus M Wörtern; der (langsame) Externspeicher (Hintergrundspeicher) hat unbeschränkte Größe. Es gibt spezielle *E/A-Operationen*, die B aufeinanderfolgende Wörter zwischen langsamem und schnellem Speicher hin- und hertransportieren. Wenn z. B. der Externspeicher eine Festplatte ist, wäre M die Größe des Hauptspeichers, und B wäre eine Blockgröße für die Datenübertragung zwischen Hauptspeicher und Platte, die einen guten Kompromiss zwischen der hohen Wartezeit (Latenzzeit) und der großen Bandweite bei einer Übertragung von einer Speicherart auf die andere darstellt. Beim aktuellen Stand der Technik sind $M = 2$ GByte und $B = 2$ MByte realistische Werte. Ein E/A-Schritt würde dann etwa 10 ms dauern, was $2 \cdot 10^7$ Taktzyklen eines 2GHz-Prozessors entspricht. Mit anderen Festlegungen für die Parameter M und B könnten wir den (kleineren) Unterschied in den Zugriffszeiten zwischen einem Hardware-Cachespeicher und dem Hauptspeicher modellieren.

2.2.2 Parallelverarbeitung

In modernen Rechnern finden wir viele Arten von Parallelverarbeitung vor. Viele Prozessoren weisen *SIMD*-Register mit einer Breite zwischen 128 und 256 Bits auf, die die parallele Ausführung eines Befehls auf einer ganzen Reihe von Datenobjekten ermöglicht („*single instruction – multiple data*“).

Simultanes Multi-Threading erlaubt es Prozessoren, ihre Ressourcen besser auszulasten, indem mehrere Threads (Teilprozesse) gleichzeitig auf einem Prozessorkern ausgeführt werden. Sogar auf mobilen Geräten gibt es oft Mehrkernprozessoren, die unabhängig voneinander Programme ausführen können, und die meisten Server haben mehrere solcher Mehrkernprozessoren, die auf einen *gemeinsamen Speicher* zugreifen können.

Coprozessoren, insbesondere solche, die in der Computergrafik Verwendung finden, weisen noch mehr parallel arbeitende Komponenten auf einem einzigen Chip auf. Hochleistungsrechner bestehen aus mehreren Server-Systemen, die durch ein eigenes schnelles Netzwerk miteinander verbunden sind. Schließlich gibt es die Möglichkeit, dass Computer aller Art über ein Netzwerk (das Internet, Funk-Netzwerke

usw.) eher lose verbunden sind und so ein *verteiltes System* bilden, in dem Millionen von Knoten zusammenarbeiten. Es ist klar, dass kein einzelnes einfaches Modell genügt, um parallele Programme zu beschreiben, die auf diesen vielen Ebenen von Parallelität ablaufen. In diesem Buch werden wir uns daher darauf beschränken, gelegentlich (ohne formale Argumentation) zu erläutern, weshalb ein bestimmter sequentieller Algorithmus vielleicht besser oder weniger gut für die parallele Ausführung umgearbeitet werden kann. Beispielsweise könnte man bei den Algorithmen für Langzahlarithmetik in Kap. 1 auch SIMD-Instruktionen einsetzen.

2.3 Pseudocode

Das RAM-Modell, eine Abstraktion und Vereinfachung der Maschinenprogramme, die auf Mikroprozessoren ausgeführt werden, soll eine präzise Definition des Begriffs „Rechenzeit“ ermöglichen. Zur Formulierung komplexer Algorithmen ist es jedoch wegen seiner Einfachheit nicht geeignet, weil die entsprechenden RAM-Programme viel zu lang und schwer lesbar wären. Stattdessen werden wir unsere Algorithmen in *Pseudocode* formulieren, der eine Abstraktion und Vereinfachung von imperativen Programmiersprachen wie C, C++, Java, C# oder Pascal darstellt, kombiniert mit großzügiger Benutzung von mathematischer Notation. Wir beschreiben nun Konventionen für die in diesem Buch verwendete Pseudocode-Notation und leiten ein Zeitmessmodell für Pseudocodeprogramme ab. Dieses ist sehr einfach: *Elementare Pseudocode-Befehle benötigen konstante Zeit; Prozedur- und Funktionsaufrufe benötigen konstante Zeit plus die Zeit für die Ausführung ihres Rumpfes*. Wir zeigen, dass dies gerechtfertigt ist, indem wir skizzieren, wie man Pseudocode in äquivalenten RAM-Code übersetzen kann, der dieses Zeitverhalten aufweist. Wir erklären diese Übersetzung nur so weit, dass das Zeitmessmodell verständlich wird. Über Compileroptimierungstechniken etwa muss man sich hier keine Gedanken machen, weil konstante Faktoren in unserer Theorie keine Rolle spielen. Der Leser mag sich auch dafür entscheiden, die folgenden Absätze zu überblättern und das Zeitmessmodell für Pseudocodeprogramme als ein Axiom zu akzeptieren. Die hier verwendete Pseudocode-Syntax ähnelt der Syntax von Pascal [110], weil diese Notation aus typographischer Sicht angenehmer für ein Buch erscheint als die wohlbekannte Syntax von C und den daraus durch Weiterentwicklung entstandenen Sprachen C++ und Java.

2.3.1 Variablen and elementare Datentypen

Eine *Variablendeklaration* „ $v = x : T$ “ stellt eine Variable v vom Typ T bereit und initialisiert sie mit dem Wert x . Beispielsweise erzeugt „ $answer = 42 : \mathbb{N}$ “ eine Variable $answer$, die nichtnegative ganzzahlige Werte annehmen kann, und initialisiert sie mit dem Wert 42. Der Typ einer Variablen wird in der Deklaration eventuell nicht genannt, wenn er aus dem Zusammenhang hervorgeht. Als Typen kommen elementare Typen wie integer (ganze Zahlen), Boolean (Boolesche Werte) oder Zeiger bzw.

Referenzen oder zusammengesetzte Typen in Frage. Dabei gibt es vordefinierte zusammengesetzte Typen wie Arrays und anwendungsspezifische Klassen (siehe unten). Wenn der Typ einer Variablen für die Diskussion irrelevant ist, benutzen wir den unspezifizierte Typ *Element* als Platzhalter für einen beliebigen Typ. Die numerischen Typen werden manchmal um die Werte $-\infty$ und ∞ erweitert, wenn dies bequem ist. Ähnlich erweitern wir manchmal Typen um einen undefinierten Wert (bezeichnet mit dem Symbol \perp), der von den „eentlichen“ Objekten vom Typ T unterscheidbar sein soll. Besonders bei Zeigertypen ist ein undefinierter Wert hilfreich. Werte des Zeigertyps „**Pointer to T** “ sind „Griffe“ (engl.: *handle*) für Objekte vom Typ T . Im RAM-Modell ist ein solcher Griff einfach der Index (die „Adresse“) der ersten Zelle eines Speichersegments, in dem ein Objekt vom Typ T gespeichert ist.

Eine Deklaration „ $a : \text{Array } [i..j] \text{ of } T$ “ stellt ein Array a bereit, das aus $j - i + 1$ Einträgen vom Typ T besteht, die in $a[i]$, $a[i + 1]$, \dots , $a[j]$ gespeichert sind. Arrays werden als zusammenhängende Speichersegmente realisiert. Um den in $a[k]$ abgelegten Eintrag zu finden, genügt es, die Startadresse von a und die Größe eines Objektes vom Typ T zu kennen. Wenn beispielsweise Register R_a die Startadresse des Arrays $a[0..k]$ enthält und R_i den Index 42, und wenn jeder Eintrag ein Speicherwort ist, dann lädt die Befehlsfolge „ $R_1 := R_a + R_i$; $R_2 := S[R_1]$ “ den Inhalt von $a[42]$ in Register R_2 . Die Größe eines Arrays wird festgelegt, wenn die Deklaration ausgeführt wird; solche Arrays heißen *statisch*. In Abschnitt 3.2 werden wir sehen, wie man *dynamische Arrays* implementiert, die während der Programmausführung wachsen und schrumpfen können.

Eine Deklaration „ $c : \text{Class } age : \mathbb{N}, income : \mathbb{N} \text{ end}$ “ stellt eine Variable c bereit, deren Werte Paare von ganzen Zahlen sind. Die Komponenten von c werden mit $c.age$ und $c.income$ angesprochen. Für eine Variable c liefert **addressof** c einen Griff für c (d. h. die Adresse von c). Wenn p eine Variable vom passenden Zeigertyp ist, dann können wir mit $p := \text{addressof } c$ diesen Griff in p speichern; mit $*p$ erhalten wir das Objekt c zurück. Die beiden „Felder“ von c können dann auch durch $p \rightarrow age$ und $p \rightarrow income$ angesprochen werden. Die alternative Schreibweise $(*p).age$ und $(*p).income$ ist möglich, aber ungebräuchlich.

Arrays und Objekte, auf die mit Zeigern verwiesen wird, können mit den Anweisungen **allocate** und **dispose** bereitgestellt und mit einem Namen versehen bzw. wieder freigegeben werden. Beispielsweise stellt die Anweisung $p := \text{allocate Array } [1..n] \text{ of } T$ ein Array von n Objekten vom Typ T bereit, d. h., es wird ein zusammenhängendes Speichersegment reserviert, dessen Länge für genau n Objekte vom Typ T ausreicht, und die Variable p erhält als Wert den Griff für das Array (die Startadresse dieses Speichersegments). Die Anweisung **dispose** p gibt den Speicherbereich frei und macht ihn damit für anderweitige Benutzung verfügbar. Mit den Operationen **allocate** und **dispose** können wir das Array S der RAM-Speicherzellen in disjunkte Stücke zerlegen, auf die man individuell zugreifen kann. Die beiden Funktionen können so implementiert werden, dass sie nur konstante Zeit benötigen, zum Beispiel auf die folgende extrem einfache Weise: Die Adresse der ersten freien Speicherzelle in S wird in einer speziellen Variablen *free* gehalten. Ein Aufruf von **allocate** reserviert einen Speicherabschnitt, der bei *free* beginnt, und erhöht *free* um

dem Umfang des reservierten Abschnitts. Ein Aufruf von **dispose** hat keinen Effekt. Diese Implementierung ist zeiteffizient, aber nicht speicherplatzeffizient: Zwar benötigt jeder Aufruf von **allocate** oder **dispose** nur konstante Zeit, aber der gesamte Speicherplatzverbrauch ist die Summe der Längen aller jemals reservierten Segmente und nicht der maximale zu einem Zeitpunkt benötigte (d. h. reservierte, aber noch nicht freigegebene) Platz. Ob jede beliebige Folge von **allocate**- und **dispose**-Operationen zugleich speicherplatzeffizient und mit konstantem Zeitaufwand für jede Operation realisiert werden kann, ist ein offenes Problem. Jedoch lassen sich für alle Algorithmen, die in diesem Buch vorgestellt werden, **allocate** und **dispose** zugleich zeit- und platzeffizient realisieren.

Einige zusammengesetzte Datentypen werden wir aus der Mathematik übernehmen. Insbesondere verwenden wir Tupel, (endliche) Folgen und Mengen. (Geordnete) *Paare*, *Tripel* und andere *Tupel* werden in Klammern geschrieben, wie in $(3, 1)$, $(3, 1, 4)$ und $(3, 1, 4, 1, 5)$. Weil sie nur eine im Typ festgelegte Anzahl von Komponenten enthalten, können Operationen auf Tupeln auf offensichtliche Weise in Operationen auf diesen Komponenten zerlegt werden. Eine *Folge*, geschrieben mit spitzen Klammern, speichert Objekte eines Typs in einer spezifizierten Reihenfolge; dabei legt der Typ die Anzahl der Einträge nicht fest. Beispielsweise deklariert die Anweisung „ $s = \langle 3, 1, 4, 1 \rangle : \text{Sequence of } \mathbb{Z}$ “ eine Folge s von ganzen Zahlen und initialisiert sie mit der Folge $\langle 3, 1, 4, 1 \rangle$ der Zahlen 3, 1, 4 und 1, in dieser Reihenfolge. Die leere Folge wird als $\langle \rangle$ geschrieben. Folgen sind eine natürliche Abstraktion vieler Datenstrukturen wie Dateien (Files), Strings (Zeichenreihen), Listen, Stapel (Stacks) und Warteschlangen (Queues). In Kap. 3 werden wir viele verschiedene Möglichkeiten betrachten, Folgen darzustellen. In späteren Kapiteln werden wir Folgen als mathematische Abstraktion vielfältig benutzen, ohne uns weiter um Implementierungsdetails zu kümmern.

Mengen spielen eine wichtige Rolle in mathematischen Überlegungen; wir werden sie auch in unserem Pseudocode benutzen. Es werden also Deklarationen wie „ $M = \{3, 1, 4\} : \text{Set of } \mathbb{N}$ “ vorkommen, die analog zu Deklarationen von Arrays oder Folgen zu verstehen sind. Die Implementierung des Datentyps „Menge“ erfolgt meist über Folgen.

2.3.2 Anweisungen

Die einfachste Anweisung ist eine *Zuweisung* $x := E$, wobei x eine Variable und E ein (in üblicher Weise aus Operationen aufgebauter) Ausdruck ist. Eine Zuweisung lässt sich leicht in eine Folge von RAM-Befehlen konstanter Länge transformieren. Beispielsweise wird die Anweisung $a := a + b \cdot c$ in „ $R_1 := R_b * R_c; R_a := R_a + R_1$ “ übersetzt, wobei R_a , R_b und R_c die Register sind, die a , b bzw. c enthalten. Aus der Programmiersprache C leihen wir uns die Abkürzungen $++$ und $--$ für das Erhöhen und das Erniedrigen einer Variablen um 1. Wie benutzen auch die simultane Zuweisung an mehrere Variablen. Wenn beispielsweise a und b Variable desselben Typs sind, vertauscht „ $(a, b) := (b, a)$ “ die Inhalte von a und b (engl.: *swap*).

Die bedingte Anweisung „**if** C **then** I **else** J “, wobei C ein Boolescher Ausdruck ist und I und J Anweisungen sind, wird in die Befehlsfolge

eval(C); JZ *sElse*, *R_C*; *trans(I)*; J *sEnd*; *trans(J)*

übersetzt. Dabei haben die einzelnen Teile folgende Bedeutung: *eval(C)* ist eine Befehlsfolge, die den Ausdruck *C* auswertet und das Ergebnis in Register *R_C* ablegt; *trans(I)* ist eine Befehlsfolge, die die Anweisung *I* implementiert; *trans(J)* implementiert die Anweisung *J*; *sElse* ist die Nummer des ersten Befehls in *trans(J)*; *sEnd* schließlich ist die Nummer des Befehls, der auf *trans(J)* folgt. Diese Folge berechnet zunächst den Wert von *C*. Wenn dies *false* (= 0) liefert, springt das RAM-Programm zum ersten Befehl der Übersetzung von *J*. Wenn das Ergebnis hingegen *true* (= 1) ist, führt das Programm die Übersetzung von *I* aus und springt nach Erreichen des Endes dieses Programmsegments zum ersten Befehl, der auf die Übersetzung von *J* folgt. Die Anweisung „*if C then I*“ ist eine Abkürzung für „*if C then I else ;*“, d. h. eine bedingte Anweisung mit leerem else-Teil.

Aus einzelnen Anweisungen kann man durch Aneinanderreihung eine Gruppe *I₁; I₂; ...; I_r* von Anweisungen erzeugen, die man auch wieder als eine (komplexere) Anweisung auffassen kann. Bei der Übersetzung in ein Maschinenprogramm werden die Übersetzungen der einzelnen Anweisungen hintereinandergesetzt.

Unsere Pseudocode-Notation für Programme ist für einen menschlichen Leser gedacht und hat daher eine nicht ganz so strenge Syntax wie Programmiersprachen. Insbesondere werden wir Anweisungen durch Einrückung zu Gruppen zusammenfassen und dadurch viele der Klammern vermeiden, die sich etwa in C-Programmen finden. Höhere Programmiersprachen wie C stellen einen Kompromiss dar: die Programme sollen für Menschen lesbar sein, aber auch maschinell verarbeitet werden können. Wir benutzen Klammern zur Gliederung von Anweisungen nur, wenn das Programm sonst mehrdeutig wäre. Genauso kann der Beginn einer neuen Zeile anstelle eines Semikolons benutzt werden, um zwei Anweisungen zu trennen.

Eine Schleife „*repeat I until C*“ wird in *trans(I)*; *eval(C)*; JZ *sI*, *R_C* übersetzt, wobei *sI* die Nummer des ersten Befehls in *trans(I)* ist. Wir werden viele andere Arten von Schleifen benutzen, die als Abkürzungen für repeat-Schleifen angesehen werden können. In der folgenden Liste sind für einige Beispiele links die Abkürzungen angegeben und rechts die jeweiligen vollen Formulierungen:

while <i>C</i> do <i>I</i>	if <i>C</i> then repeat <i>I</i> until $\neg C$
for <i>i</i> := <i>a</i> to <i>b</i> do <i>I</i>	<i>i</i> := <i>a</i> ; while <i>i</i> ≤ <i>b</i> do <i>I</i> ; <i>i</i> ++
for <i>i</i> := <i>a</i> to ∞ while <i>C</i> do <i>I</i>	<i>i</i> := <i>a</i> ; while <i>C</i> do <i>I</i> ; <i>i</i> ++
foreach <i>e</i> ∈ <i>s</i> do <i>I</i>	for <i>i</i> := 1 to <i>s</i> do <i>e</i> := <i>s</i> [<i>i</i>]; <i>I</i>

Bei der Übersetzung von Schleifen in RAM-Code sind viele maschinennahe Optimierungen möglich, die wir aber nicht berücksichtigen. Für unsere Zwecke ist nur von Bedeutung, dass die Rechenzeit für die Ausführung einer Schleife die Summe der Rechenzeiten für die einzelnen Durchläufe ist, wobei natürlich die Zeit für die Auswertung der Bedingungen mit berücksichtigt werden muss.

2.3.3 Prozeduren und Funktionen

Ein Unterprogramm mit Namen *foo* wird wie folgt deklariert: „**Procedure** *foo(D)* *I*“. Dabei ist *I* der Prozedurrumpf, und *D* ist eine Liste von Variablendeklarationen,

die die „formalen“ Parameter von *foo* spezifizieren. Ein Aufruf von *foo* hat die Form *foo(P)*, wobei *P* eine Liste von „aktuellen“⁴ Parametern oder Argumenten ist, die genauso lang wie die Liste der Variablendeklarationen ist. Aktuelle Parameter sind entweder Wertparameter (engl.: *pass by value*) oder Referenzparameter (engl.: *pass by reference*). Wenn nichts anderes gesagt ist, nehmen wir an, dass elementare Objekte wie ganze Zahlen oder Boolesche Variablen als Wert übergeben werden, komplexe Objekte wie Arrays dagegen als Referenz. Diese Konventionen entsprechen denen, die in C benutzt werden; sie stellen sicher, dass die Parameterübergabe konstante Zeit benötigt. Die Semantik der Parameterübergabe ist dabei folgende: Ist *x* ein Wertparameter vom Typ *T*, dann muss der entsprechende aktuelle Parameter im Aufruf ein Ausdruck *E* sein, der einen Wert vom Typ *T* liefert. Die Parameterübergabe ist dann äquivalent zur Deklaration einer lokalen Variablen mit Namen *x*, die mit dem Wert von *E* initialisiert wird. Ist *x* dagegen ein Referenzparameter vom Typ *T*, dann muss der entsprechende aktuelle Parameter im Aufruf eine Variable desselben Typs sein; während der Ausführung des Prozedurrumpfes ist *x* einfach ein alternativer Name für diese Variable.

Ebenso wie bei Variablendeklarationen lassen wir mitunter die Angabe des Typs bei Parameterdekларationen weg, wenn er unwesentlich oder aus dem Zusammenhang ersichtlich ist. Manchmal werden Parameter auch durch Verwendung von mathematischer Notation nur implizit deklariert. Beispielsweise deklariert „**Procedure** *bar*((*a*₁, ..., *a*_{*n*})) *I*“ eine Prozedur, deren Argument eine Folge von Elementen eines nicht weiter bezeichneten Typs ist.

Im Prozedurrumpf darf die spezielle Anweisung **return** vorkommen, die die Ausführung des Schleifentrumpfes beendet und mit der Ausführung der ersten Anweisung nach der Aufrufstelle fortfährt. Wird bei der Ausführung das Ende des Prozedurrumpfes erreicht, hat dies denselben Effekt.

Die meisten Prozeduraufrufe können übersetzt werden, indem einfach der Prozedurrumpf für den Aufruf eingesetzt wird und Maßnahmen für die Parameterübergabe vorgesehen werden. Dieses Vorgehen nennt man „Inline-Aufruf“ (engl.: *inlining*). Die Übergabe eines Wertparameters *E* für einen formalen Parameter *x* : *T* wird durch eine Folge von Befehlen realisiert, die den Ausdruck *E* auswerten und das Resultat der lokalen Variablen *x* zuweisen. Bei einem formalen Referenzparameter *x* : *T* geschieht die Übergabe dadurch, dass die lokale Variable *x* den Typ **Pointer to T** erhält und alle Vorkommen von *x* im Prozedurrumpf durch *(*x)* ersetzt werden. Zudem muss vor Betreten des Prozedurrumpfes die Zuweisung *x* := **addressof** *y* ausgeführt werden, wobei *y* der aktuelle Parameter ist. Die Benutzung von Inline-Aufrufen gibt dem Compiler viele Optimierungsmöglichkeiten, so dass dieses Vorgehen zum effizientesten Code führt, wenn es sich um kleine Prozeduren oder um Prozeduren handelt, die nur von einer Stelle aus aufgerufen werden.

Funktionen ähneln Prozeduren, nur ist bei ihnen vorgesehen, dass die **return**-Anweisung einen Wert zurückgibt. In Abb. 2.2 findet man die Deklaration einer

⁴ Anm: d. Ü.: Der Ausdruck „aktueller Parameter“ in diesem Zusammenhang ist eine seit langem eingebürgerte Fehlübersetzung von „*actual parameter*“; dies heißt eigentlich „konkreter/tatsächlicher Parameter“.

Function $factorial(n) : \mathbb{N}$

if $n = 0$ **then return** 1 **else return** $n \cdot factorial(n - 1)$

```

factorial :                                // erster Befehl von factorial
Rarg := RS[Rr - 1]                        // lade n in Register Rarg
JZ thenCase, Rarg                          // springe zum then-Fall, falls n = 0
RS[Rr] := aRecCall                          // else-Fall; Rückkehradresse für rek. Aufruf
RS[Rr + 1] := Rarg - 1                    // Parameter ist n - 1
Rr := Rr + 2                              // erhöhe Stackzeiger
J factorial                                // starte rekursiven Aufruf
aRecCall :                                // Rückkehradresse für rek. Aufruf
Rresult := RS[Rr - 1] * Rresult          // speichere n * factorial(n - 1) ins Resultatregister
J return                                  // gehe zur Rücksprung-Vorbereitung
thenCase :                                // Code für then-Fall
Rresult := 1                              // speichere 1 ins Resultatregister
return :                                  // Code für Rücksprung
Rr := Rr - 2                              // gib Aktivierungssatz frei
J RS[Rr]                                  // springe zur Rückkehradresse

```

Abb. 2.2. Eine rekursive Funktion *factorial* zur Berechnung der Fakultätsfunktion und der entsprechende RAM-Code. Der RAM-Code gibt das Resultat $n!$ in Register R_{result} zurück.

R_r	
	3
	aRecCall
	4
	aRecCall
	5
	afterCall

Abb. 2.3. Rekursionsstack eines Aufrufs *factorial*(5), wenn die Rekursion den Aufruf *factorial*(3) erreicht hat.

Funktion *factorial*, die auf Eingabe n das Ergebnis $n!$ zurückgibt, sowie ihre Übersetzung in RAM-Code. Die Inline-Ersetzung funktioniert nicht für *rekursive* Prozeduren und Funktionen, die sich (wie *factorial*) direkt oder indirekt selber aufrufen – die Code-Ersetzung würde nie enden. Um rekursive Unterprogramme in RAM-Code zu realisieren, benötigt man den Begriff des *Rekursionsstacks* (oder Rekursionsstapels). Man benutzt explizite Aufrufe und den Rekursionsstack auch für umfangreiche Unterprogramme, die von mehreren Stellen im Programm aus aufgerufen werden, weil dabei die Inline-Ersetzung den Code zu sehr aufblähen würde. Der Rekursionsstack, bezeichnet mit RS , wird in einem reservierten Teil des Speichers angelegt. Er enthält eine Folge von „Aktivierungssätzen“ (engl.: *activation record*), einen für jeden aktiven Unterprogrammaufruf. Ein spezielles Register R_r zeigt immer auf den ersten freien Platz oberhalb dieses Stacks. Der Aktivierungssatz für ein Unterprogramm mit k Parametern und ℓ lokalen Variablen hat Größe $1 + k + \ell$. Die erste Position enthält die „Rücksprungadresse“, d. h. die Nummer der Programmzeile, bei der die

Ausführung fortgesetzt werden soll, wenn der Aufruf beendet ist; die nächsten k Positionen sind für die Parameter reserviert, die restlichen ℓ Positionen für die lokalen Variablen. Ein Unterprogrammaufruf wird nun im RAM-Code wie folgt realisiert: Zuerst schreibt die aufrufende Prozedur *caller* die Rücksprungadresse und die aktuellen Parameter auf den Stack, erhöht den Wert in R_r um $1 + k$ und springt zur ersten Programmzeile des Unterprogramms *called*. Das aufgerufene Unterprogramm reserviert als Erstes durch Erhöhen von R_r Platz für seine ℓ lokalen Variablen. Dann wird der Rumpf von *called* abgearbeitet. Wird während dieser Abarbeitung der i -te formale Parameter ($0 \leq i < k$) angesprochen, so wird auf die Zelle $RS[R_r - \ell - k + i]$ zugegriffen; ein Zugriff auf die j -te lokale Variable ($0 \leq j < \ell$) wird durch einen Zugriff auf $RS[R_r - \ell + j]$ realisiert. Eine **return**-Anweisung in *called* wird in eine Folge von RAM-Befehlen umgesetzt, die R_r um $1 + k + \ell$ erniedrigt (*called* kennt natürlich die Zahlen k und ℓ) und dann zur in $RS[R_r]$ gespeicherten Rückkehradresse springt. Hierdurch wird die Ausführung der Prozedur *caller* fortgesetzt. Man beachte, dass mit dieser Methode Rekursion kein Problem mehr darstellt, denn jeder Aufruf („Inkarnation“) eines Unterprogramms hat seinen eigenen Bereich im Stack für seine Parameter und lokale Variablen. Abbildung 2.3 zeigt den Inhalt des Rekursionsstacks für einen Aufruf *factorial*(5) in dem Moment, in dem die Rekursion den Aufruf *factorial*(3) erreicht hat. Die Marke *afterCall* steht für die Nummer der Programmzeile, die auf den Aufruf *factorial*(5) folgt; *aRecCall* wird in Abb. 2.2 definiert. Für diesen speziellen Fall erlauben wir in Erweiterung des ganz einfachen RAM-Befehlssatzes auch $RS[R_r]$ als Ziel eines unbedingten Sprungs.

Aufgabe 2.5 (Sieb des Eratosthenes). Übersetzen Sie den folgenden Pseudocode, der alle Primzahlen bis zur Zahl n findet und ausgibt, in RAM-Code. (Die letzte Zeile, in der die eingerahmte Zahl als Wert ausgegeben wird, braucht nicht übersetzt zu werden.) Zeigen Sie zunächst, dass der Algorithmus korrekt ist.

```

a = ⟨1, ..., 1⟩ : Array [2..n] of {0, 1}
// Referenzparameter; am Ende: a[i] = 1 ⇔ i ist Primzahl
for i := 2 to ⌊√n⌋ do
  if a[i] then for j := 2i to n step i do a[j] := 0
  // Wenn a[i] = 1, ist i prim, Vielfache von i dagegen nicht
for i := 2 to n do if a[i] then output(“⌊i⌋ ist Primzahl”)

```

2.3.4 Objektorientierung

Wir werden eine einfache Form der objektorientierten Programmierung benutzen, um die Spezifikation (engl.: *interface*) und die Implementierung von Datenstrukturen zu trennen. Die entsprechende Notation soll hier anhand eines Beispiels erklärt werden. Die Definition

```

Class Complex(x, y : Number) of Number
  re = x : Number
  im = y : Number
  Function abs : ℝ return √re2 + im2
  Function add(c' : Complex) : Complex return Complex(re + c'.re, im + c'.im)

```

stellt eine (partielle) Implementierung eines Zahlentyps für komplexe Zahlen dar, der beliebige elementare Zahlentypen wie \mathbb{Z} , \mathbb{Q} und \mathbb{R} als Real- und Imaginärteil benutzen kann. („Complex“ ist ein Beispiel dafür, dass wir unseren Klassen oft Namen geben werden, die mit einem Großbuchstaben beginnen.) Real- und Imaginärteil werden in den *Instanzvariablen* *re* bzw. *im* gehalten. Die Deklaration „*c* : *Complex*(2, 3) **of** \mathbb{R} “ erzeugt eine komplexe Zahl *c*, die mit $2 + 3i$ initialisiert ist, wobei *i* die imaginäre Einheit ist. Mit *c.im* erhält man den Imaginärteil von *c* und mit dem Methodenaufruf *c.abs* den Absolutbetrag von *c* – einen reellen Wert.

Die Angabe des Typs nach dem **of** in der Variablendeklaration ermöglicht es, Klassen in ähnlicher Weise zu parametrisieren wie mit dem Template-Mechanismus in C++ oder mit den generischen Typen in Java. Man beachte, dass mit dieser Notation die früher erwähnten Deklarationen „*Set of Element*“ und „*Sequence of Element*“ ganz gewöhnliche (parametrisierte) Klassen definieren. Objekte einer Klasse werden initialisiert, indem den Instanzvariablen die in der Variablendeklaration angegebenen Werte zugewiesen werden.

2.4 Erstellung korrekter Algorithmen und Programme

Ein Algorithmus ist eine allgemeine Methode, um Probleme einer bestimmten Art zu lösen. Wir beschreiben Algorithmen in natürlicher Sprache und mit Hilfe von mathematischer Notation. Algorithmen an sich können nicht von einem Computer ausgeführt werden. Die Formulierung eines Algorithmus in einer Programmiersprache nennt man Programm. Korrekte Algorithmen zu entwerfen und korrekte Algorithmen in korrekte Programme zu übertragen sind nichttriviale Aufgaben, bei denen Fehler auftreten können. In diesem Abschnitt befassen wir uns mit Zusicherungen und Invarianten, zwei nützlichen Konzepten, die die Erstellung korrekter Algorithmen und Programme unterstützen.

Function *power*(*a* : \mathbb{R} ; *n*₀ : \mathbb{Z}) : \mathbb{R}

```

assert n0 ≥ 0           // negative Exponenten können nicht bearbeitet werden
p = a :  $\mathbb{R}$ ;   r = 1 :  $\mathbb{R}$ ;   n = n0 :  $\mathbb{N}$            // es gilt  $p^n r = a^{n_0}$ 
while n > 0 do
  invariant  $p^n r = a^{n_0}$ 
  if n ist ungerade then n --; r := r · p // zwischen Zuweisungen ist Invariante verletzt
  else (n, p) := (n/2, p · p) // parallele Zuweisung erhält Invariante aufrecht
  assert  $r = a^{n_0}$            // Folgerung aus der Invarianten und n = 0
return r

```

Abb. 2.4. Ein Algorithmus zur Berechnung nichtnegativer ganzzahliger Potenzen von reellen Zahlen.

2.4.1 Zusicherungen und Invarianten

Zusicherungen und *Invarianten* beschreiben Eigenschaften des Programmzustandes, d. h. Eigenschaften von einzelnen Variablen und Beziehungen zwischen den Werten verschiedener Variablen. Typische solche Eigenschaften sind etwa, dass ein Zeiger einen bestimmten Wert hat, dass eine ganze Zahl nicht negativ ist, dass eine Liste nicht leer ist oder dass der Wert einer Variablen *length* gleich der Länge einer bestimmten Liste *L* ist. Abbildung 2.4 zeigt ein Beispiel für den Gebrauch von Zusicherungen und Invarianten in einer Funktion $power(a, n_0)$, die für eine gegebene reelle Zahl *a* und eine nichtnegative ganze Zahl n_0 den Wert a^{n_0} berechnet.

Wir beginnen mit der Zusicherung **assert** $n_0 \geq 0$. Diese stellt fest, dass die Funktion erwartet, dass die ganze Zahl n_0 in der Eingabe nicht negativ ist. Über das Verhalten des Programms im Fall, dass diese Zusicherung nicht erfüllt ist, wird nichts gesagt. Diese Zusicherung wird daher die *Vorbedingung* des Programms genannt. Zur guten Programmierpraxis gehört es, die Vorbedingung zu überprüfen, d. h. Code zu schreiben, der sie überprüft und einen Fehler anzeigt, wenn sie verletzt ist. Wenn die Vorbedingung erfüllt ist (und das Programm korrekt ist), dann gilt bei Beendigung des Programms eine *Nachbedingung*. In unserem Beispiel ist dies die Zusicherung, dass $r = a^{n_0}$ gilt.

Vor- und Nachbedingungen kann man als eine Art *Vertrag* („*design by contract*“) zwischen dem aufrufenden Programm und dem aufgerufenen Unterprogramm auffassen: Der Aufrufer muss sicherstellen, dass die beim Aufruf übergebenen Parameter die Vorbedingung erfüllen; das Unterprogramm garantiert, dass dann ein Resultat erzeugt wird, das die Nachbedingung erfüllt.

Um die Programme nicht ausufern zu lassen, werden wir Zusicherungen sparsam verwenden und oft einfach annehmen, dass gewisse Bedingungen gelten, die sich „offensichtlich“ aus der textuellen Beschreibung des Algorithmus ergeben. Deutlich ausgefeiltere Systeme von Zusicherungen werden für sicherheitskritische Programme oder für die formale Verifikation von Programmen benötigt.

Vor- und Nachbedingungen sind Zusicherungen, die den Zustand eines Programms zu Beginn und am Ende der Ausführung beschreiben. Wir müssen aber auch Eigenschaften an Stellen während der Ausführung betrachten. Oft steht und fällt der Korrektheitsbeweis für einen Algorithmus damit, dass man einige besonders wichtige Konsistenzeigenschaften identifiziert und formuliert, die immer gelten, wenn eine bestimmte Stelle im Programm durchlaufen wird. Solche Eigenschaften heißen *Invarianten*. Schleifeninvarianten und Datenstruktur-Invarianten sind dabei besonders wichtig.

2.4.2 Schleifeninvarianten

Eine *Schleifeninvariante* gilt vor und nach jedem Schleifendurchlauf. In unserem Beispiel behaupten wir, dass vor jedem Schleifendurchlauf die Gleichung $p^n r = a^{n_0}$ gilt. Sie ist vor dem ersten Durchlauf richtig: Dies wird durch die Initialisierung der lokalen Variablen sichergestellt. Tatsächlich legen Invarianten oft fest, wie die Variablen initialisiert werden müssen. Nun nehmen wir an, dass vor einer Ausführung des

Schleifenrumpfes die Invariante gilt, und dass $n > 0$ ist. Wenn n ungerade ist, verringern wir n um 1 und multiplizieren r mit p . Diese Operationen stellen die Invariante wieder her. (Man beachte, dass sie zwischen den Zuweisungen verletzt ist!) Wenn n gerade ist, halbieren wir n und quadrieren p ; auch in diesem Fall ist die Invariante wieder hergestellt. Wenn die Schleife endet, gilt $p^n r = a^{n_0}$ wegen der Invarianten und $n = 0$ wegen der Schleifenendebedingung. Daraus folgt $r = a^{n_0}$, und wir haben die Nachbedingung bewiesen.

Der Algorithmus in Abb. 2.4 und viele andere Algorithmen in diesem Buch haben eine recht einfache Struktur. Einige Variablen werden deklariert und initialisiert, um die Schleifeninvariante gültig zu machen. Dann folgt eine Hauptschleife, die den Zustand des Programms verändert. Wenn die Schleife endet, folgt aus der Gültigkeit der Schleifeninvarianten zusammen mit der Abbruchbedingung der Schleife, dass das richtige Ergebnis berechnet wurde. Die Schleifeninvariante trägt dadurch entscheidend dazu bei, dass man versteht, weshalb ein Programm korrekt arbeitet. Wenn man erst einmal die „richtige“ Schleifeninvariante aufgestellt hat, muss man nur noch nachprüfen, dass sie am Anfang gilt, und dass die Ausführung eines Durchlaufs ihre Gültigkeit aufrechterhält. Dies ist natürlich besonders leicht, wenn wie im obigen Beispiel der Schleifenrumpf nur aus einer kleinen Anzahl von Anweisungen besteht.

2.4.3 Datenstruktur-Invarianten

Bei komplexeren Programmen wird man den Zustand in Objekte einkapseln, wobei dann die Konsistenz der Darstellung wieder durch Invarianten sichergestellt wird. Solche *Datenstruktur-Invarianten* werden zusammen mit dem Datentyp (d. h. der entsprechenden Klasse) deklariert. Sie gelten unmittelbar nach der Konstruktion eines Objektes, und sie sind Vorbedingung und Nachbedingung für alle Methoden der Klasse. Beispielsweise werden wir diskutieren, wie man Mengen durch geordnete Arrays darstellen kann. Die Datenstruktur-Invariante wird dabei aus folgenden Aussagen bestehen: Die Datenstruktur benutzt ein Array a und eine ganze Zahl n ; die Zahl n ist die Länge von a ; die Menge S , die in der Datenstruktur gespeichert ist, ist genau $\{a[1], \dots, a[n]\}$; es gilt $a[1] < a[2] < \dots < a[n]$. Die Methoden der Klasse müssen diese Invariante aufrechterhalten, sie dürfen sie aber auch verwenden; beispielsweise darf die Suchmethode ausnutzen, dass das Array geordnet ist.

2.4.4 Zertifizierung von Algorithmen

Wir haben oben erwähnt, dass es zur guten Programmierpraxis gehört, Zusicherungen nachzuprüfen. Es ist nun nicht immer klar, wie dies effizient bewerkstelligt werden kann; in unsere Beispielprogramme war es einfach, die Vorbedingung zu überprüfen, aber es scheint keine einfache Methode zur Überprüfung der Nachbedingung zu geben. In vielen Situationen kann aber die *Überprüfung von Zusicherungen* dadurch vereinfacht werden, dass man *zusätzliche Information berechnet*. Diese zusätzliche Information heißt ein *Zertifikat* oder *Zeuge* (engl.: *witness*); sie soll die Überprüfung

einer Zusicherung erleichtern. Einen Algorithmus, der ein Zertifikat für die Nachbedingung berechnet, nennen wir einen *zertifizierenden Algorithmus*. Ein Beispiel soll die Idee veranschaulichen. Wir betrachten eine Funktion, deren Eingabe ein Graph $G = (V, E)$ ist. (Graphen werden in Abschnitt 2.9 definiert.) Die Aufgabe ist zu entscheiden, ob der Graph bipartit ist, d. h., ob es eine Färbung der Knoten von G mit den Farben blau und rot gibt, so dass jede Kante zwei verschiedenfarbige Knoten verbindet. Mit dieser Spezifikation liefert die Funktion als Ergebnis *true* (wenn der Graph bipartit ist) oder *false* (wenn er nicht bipartit ist). Mit dieser summarischen Ausgabe kann die Nachbedingung nicht überprüft werden. Wir können aber das Programm wie folgt erweitern: wenn es erklärt, dass G bipartit ist, liefert es auch eine 2-Färbung der Knoten des Graphen; wenn es erklärt, dass G nicht bipartit ist, liefert es auch einen Kreis ungerader Länge in G . Für das so erweiterte Programm kann die Nachbedingung leicht überprüft werden. Im ersten Fall testen wir jede Kante darauf, ob ihre Endpunkte verschiedenfarbig sind. Im zweiten Fall testen wir, ob die ausgegebene Knotenfolge, etwa v_0, v_1, \dots, v_k , tatsächlich einen Kreis in G mit ungerader Länge bildet. Die meisten Algorithmen in diesem Buch können zu zertifizierenden Versionen erweitert werden, ohne die asymptotische Rechenzeit zu erhöhen.

2.5 Ein Beispiel: Binäre Suche

Binäre Suche ist eine äußerst nützliche Technik, mit der man in einer geordneten Menge von Elementen suchen kann. Wir werden sie später sehr oft benutzen.

Das einfachste Szenario sieht folgendermaßen aus. Gegeben ist ein geordnetes Array $a[1..n]$ mit paarweise verschiedenen Einträgen, d. h. $a[1] < a[2] < \dots < a[n]$, sowie ein Element x . Wir sollen entscheiden, ob x im Array vorkommt, und den Index $k \in 1..n+1$ finden, für den $a[k-1] < x \leq a[k]$ gilt. Dabei sollen $a[0]$ und $a[n+1]$ als fiktive Einträge mit Werten $-\infty$ bzw. $+\infty$ aufgefasst werden. In Invarianten und Beweisen können wir diese fiktiven Einträge benutzen, aber im Programm darf auf sie nicht zugegriffen werden.

Binäre Suche beruht auf dem Teile-und-Herrsche-Prinzip. Wir wählen einen Index $m \in 1..n$ und vergleichen x mit $a[m]$. Falls $x = a[m]$ gilt, sind wir fertig und geben $i = m$ zurück. Im Fall $x < a[m]$ können wir uns bei der weiteren Suche auf den Teil des Arrays vor $a[m]$ beschränken, im Fall $x > a[m]$ auf den Teil des Arrays nach $a[m]$. Wir müssen natürlich genauer erklären, was es heißen soll, die Suche auf ein Teilarray zu beschränken. Wir führen zwei Indizes ℓ und r mit und sorgen dafür, dass stets die Invariante

$$0 \leq \ell < r \leq n+1 \quad \text{und} \quad a[\ell] < x < a[r] \quad (\text{I})$$

gilt. Sie gilt am Anfang mit $\ell = 0$ und $r = n+1$. Wenn ℓ und r aufeinanderfolgende Indizes sind, kann x nicht im Array vorkommen. Abbildung 2.5 zeigt das vollständige Programm.

Die Kommentare im Programm zeigen, dass der zweite Teil der Invariante stets gilt. Bezüglich des ersten Teils beobachten wir, dass die Schleife mit Werten $\ell < r$

betreten wird. Wenn $\ell + 1 = r$ gilt, endet die Schleife und die Prozedur. Andernfalls gilt $\ell + 2 \leq r$, und daraus folgt $\ell < m < r$. Daher ist m ein legaler Arrayindex, und wir können auf $a[m]$ zugreifen. Wenn nun $x = a[m]$ ist, endet die Schleife mit der korrekten Ausgabe. Andernfalls setzen wir entweder $r = m$ oder $\ell = m$; daher gilt $\ell < r$ auch am Ende der Schleife, und die Invariante bleibt erhalten.

Nun wollen wir nachweisen, dass die Schleife terminiert. Wir beobachten zunächst, dass in einem Durchlauf, der nicht der letzte ist, entweder ℓ erhöht oder r erniedrigt wird, d. h., dass $r - \ell$ auf jeden Fall abnimmt. Daher terminiert die Schleife. Wir wollen aber mehr zeigen, nämlich dass die Schleife schon nach einer logarithmischen Anzahl von Durchläufen terminiert. Hierfür betrachten wir die Größe $r - \ell - 1$. Dies ist die Anzahl der Indizes i mit $\ell < i < r$ und daher ein natürliches Maß für die Größe des aktuellen Teilproblems. Wir zeigen nun, dass sich in jedem Schleifendurchlauf außer dem letzten die Größe des Teilproblems mindestens halbiert. Wenn es sich nicht um den letzten Durchlauf handelt, verringert sich $r - \ell - 1$ auf einen Wert, der nicht größer als

$$\begin{aligned} & \max\{r - \lfloor (r + \ell)/2 \rfloor - 1, \lfloor (r + \ell)/2 \rfloor - \ell - 1\} \\ & \leq \max\{r - ((r + \ell)/2 - 1/2) - 1, (r + \ell)/2 - \ell - 1\} \\ & = \max\{(r - \ell - 1)/2, (r - \ell)/2 - 1\} = (r - \ell - 1)/2 \end{aligned}$$

ist, und wird damit mindestens halbiert. Wir beginnen mit $r - \ell - 1 = n + 1 - 0 - 1 = n$. Daher gilt nach $k \geq 0$ Durchläufen die Ungleichung $r - \ell - 1 \leq n/2^k$. Durchlauf $k + 1$ sei nun der letzte durchgeführte Durchlauf, in dem der Vergleich zwischen x und Eintrag $a[m]$ stattfindet. (Wenn die Suche erfolgreich ist, stellt sich heraus, dass $x = a[m]$ gilt, und die Suche endet. Wenn die Suche erfolglos ist, ergibt der Test in Runde $k + 2$, dass $r = \ell + 1$ gilt, und die Suche endet ohne einen weiteren Vergleich.) Dann muss nach Runde k die Ungleichung $r - \ell - 1 \geq 1$ gültig gewesen sein. Es folgt $1 \leq n/2^k$, und damit $2^k \leq n$, also $k \leq \log n$. Weil k ganzzahlig ist, kann dies zu $k \leq \lfloor \log n \rfloor$ verschärft werden.

$(\ell, r) := (0, n + 1)$

while *true* **do**

invariant (I)

// d. h. Invariante (I) gilt hier

if $\ell + 1 = r$ **then return** (“($a[\ell] < x < a[\ell + 1]$)”)

$m := \lfloor (r + \ell)/2 \rfloor$

// $\ell < m < r$

$s := \text{compare}(x, a[m])$

// -1 falls $x < a[m]$, 0 falls $x = a[m]$, $+1$ falls $x > a[m]$

if $s = 0$ **then return** (“ x steht in $a[\underline{m}]$ ”)

if $s < 0$

then $r := m$

// $a[\ell] < x < a[m] = a[r]$

else $\ell := m$

// $a[\ell] = a[m] < x < a[r]$

Abb. 2.5. Binäre Suche nach x in einem geordneten Array $a[1..n]$. (Eingetragene Zahlen werden als Werte in die Textausgabe eingebaut.)

Satz 2.3 *Binäre Suche findet ein Element in einem geordneten Array der Länge n in nicht mehr als $1 + \lfloor \log n \rfloor$ Vergleichen zwischen Einträgen. Die Rechenzeit ist $O(\log n)$.*

Aufgabe 2.6. Zeigen Sie, dass diese Schranke scharf ist, d. h., dass es für jedes n Eingaben der Größe n gibt, bei denen $1 + \lfloor \log n \rfloor$ Vergleiche benötigt werden.

Aufgabe 2.7. Formulieren Sie binäre Suche mit Zweiweg-Vergleichen, d. h., ein Vergleich liefert nur, ob $x \leq a[m]$ oder $x > a[m]$ gilt.

Als Nächstes diskutieren wir zwei wichtige Erweiterungen der binären Suche. Erstens müssen die Werte $a[i]$ nicht in einem Array gespeichert sein. Wir müssen nur $a[i]$ berechnen können, wenn i gegeben ist. Wenn beispielsweise eine streng monoton wachsende Funktion f und Argumente i und j mit $f(i) < x \leq f(j)$ gegeben sind, dann können wir binäre Suche benutzen, um den Index $k \in i + 1..j$ mit $f(k-1) < x \leq f(k)$ zu finden. In diesem Zusammenhang heißt binäre Suche oft auch „Bisektionsmethode“.

Zweitens können wir binäre Suche auf den Fall erweitern, in dem das Array unendlich lang ist. Nehmen wir an, wir hätten ein geordnetes Array $a[1..\infty]$ und möchten das $k \in \{1, 2, 3, \dots\}$ finden, das $a[k] < x \leq a[k+1]$ erfüllt. Falls x größer ist als alle Einträge im Array, darf die Prozedur auch unendlich lange rechnen. Wir gehen folgendermaßen vor: Wir vergleichen x mit $a[2^0]$, $a[2^1]$, $a[2^2]$, $a[2^3]$, \dots , bis das erste i mit $x \leq a[2^i]$ gefunden wird. Dieses Verfahren nennt man *exponentielle Suche*. Falls $i = 0$, wird $k = 1$ zurückgegeben, andernfalls wird die Suche mit gewöhnlicher binärer Suche im Array $a[2^{i-1} + 1..2^i]$ abgeschlossen.

Satz 2.4 *Die Kombination von exponentieller und binärer Suche findet x in einem unbeschränkten geordneten Array in höchstens $2 \log k + 3$ Vergleichen, wobei k durch die Bedingung $a[k-1] < x \leq a[k]$ festgelegt ist.*

Beweis. Wir benötigen $i + 1$ Vergleiche, um das kleinste i mit $x \leq a[2^i]$ zu finden, und (falls $i > 0$) anschließend maximal $\log(2^i - 2^{i-1}) + 1 = i$ Vergleiche für die binäre Suche. Zusammen sind dies $2i + 1$ Vergleiche. Da $k \geq 2^{i-1}$, gilt $i \leq 1 + \log k$, womit sich die Behauptung ergibt. \square

Binäre Suche ist ein zertifizierender Algorithmus. Sie gibt einen Index k zurück, der $a[k-1] < x \leq a[k]$ erfüllt. Falls $x = a[k]$, belegt der Index k , dass x im Array vorkommt. Falls $a[k-1] < x < a[k]$, und das Array geordnet ist, belegt der Index k , dass x nicht im Array vorkommen kann. Wenn natürlich das Array die Vorbedingung verletzt und gar nicht geordnet ist, wissen wir nichts. Es ist unmöglich, die Vorbedingung in logarithmischer Zeit zu überprüfen.

2.6 Grundlagen der Algorithmenanalyse

Die Prinzipien der Algorithmenanalyse, soweit bisher betrachtet, lassen sich wie folgt zusammenfassen. Wir abstrahieren von den Komplikationen eines realen Rechners, indem wir das vereinfachte RAM-Modell betrachten. In diesem Modell wird

die Rechenzeit dadurch gemessen, dass man die Anzahl der ausgeführten Befehle zählt. Wir vereinfachen die Analyse weiter, indem wir Eingaben nach ihrer Größe gruppieren und uns auf den schlechtesten Fall konzentrieren. Die asymptotische Notation ermöglicht es uns, konstante Faktoren und Terme kleinerer Ordnung zu ignorieren. Diese vergrößerte Sichtweise erlaubt es uns auch, obere Schranken für die Ausführungszeiten zu betrachten, solange das asymptotische Ergebnis sich nicht ändert. Insgesamt ergibt sich aus diesen Vereinfachungen, dass die Rechenzeit von Pseudocode direkt analysiert werden kann, ohne dass man das Programm in Maschinensprache übersetzen muss.

Als Nächstes stellen wir einen Satz einfacher Regeln für die Analyse von Pseudocode vor. Es sei $T_n(I)$ die maximale Ausführungszeit eines Programmstücks I auf Eingaben einer bestimmten Größe n . Die folgenden Regeln sagen uns dann, wie wir die Rechenzeit für größere Programme abschätzen können, wenn wir die Rechenzeiten für ihre Bestandteile kennen:

- Für nacheinander ausgeführte Anweisungen: $T_n(I; J) \leq T_n(I) + T_n(J)$.
- Für bedingte Anweisungen: $T_n(\text{if } C \text{ then } I \text{ else } J) = T_n(C) + \max\{T_n(I), T_n(J)\}$.
- Für Schleifen: $T_n(\text{repeat } I \text{ until } C) = \sum_{i=1}^{k(n)} T'_n(I, C, i)$, wobei $k(n)$ die maximale Anzahl der Schleifendurchläufe auf Eingaben der Größe n ist und $T'_n(I, C, i)$ die Ausführungszeit des i -ten Schleifendurchlaufs ist, inklusive des Tests C .

Unterprogrammaufrufe werden später in Abschnitt 2.6.2 betrachtet. Von den bisher angegebenen Regeln ist nur die Regel für Schleifen nicht trivial; bei ihr muss man Summen auswerten.

2.6.1 Summationen

Wir stellen hier einige Grundtechniken für die Auswertung von Summen vor. Summen tauchen bei der Analyse von Schleifen, bei der Analyse von mittleren Rechenzeiten und bei der Analyse von randomisierten Algorithmen auf.

Beispielsweise weist der Algorithmus „Einfügesortieren“, der in Abschnitt 5.1 vorgestellt wird, zwei geschachtelte Schleifen auf. Die äußere Schleife zählt die Variable i von 2 bis n hoch. Die innere Schleife wird maximal $(i - 1)$ -mal durchlaufen. Daher ist die Gesamtanzahl der Durchläufe durch die innere Schleife höchstens

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

wobei sich die zweite Gleichung aus (A.11) ergibt. Da eine Ausführung der inneren Schleife Zeit $O(1)$ benötigt, erhalten wir eine Ausführungszeit im schlechtesten Fall von $\Theta(n^2)$. Alle geschachtelten Schleifen mit einer Anzahl von Iterationen, die leicht vorhergesagt werden kann, können auf analoge Weise analysiert werden: man arbeitet „von innen nach außen“ und versucht dabei, für die Rechenzeit der jeweils aktuellen „innersten Schleife“ eine Abschätzung durch einen geschlossenen Ausdruck zu finden. Mit einfachen Umformungen wie $\sum_i ca_i = c \sum_i a_i$, $\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$

oder $\sum_{i=2}^n a_i = -a_1 + \sum_{i=1}^n a_i$ kann man die Summen oft auf eine einfache Form bringen, die man in einem Katalog von Summenformeln findet. Eine kleine Auswahl solcher Formeln ist in Anhang A bereitgestellt. Weil wir uns normalerweise nur für das asymptotische Verhalten interessieren, können wir oft darauf verzichten, exakte Formeln für die Summen zu finden, und stattdessen auf Abschätzungen zurückgreifen. Zum Beispiel könnten wir, anstatt die Summe von oben exakt auszuwerten, einfacher wie folgt abschätzen (für $n \geq 2$):

$$\begin{aligned}\sum_{i=2}^n (i-1) &\leq \sum_{i=1}^n n = n^2 = O(n^2) , \\ \sum_{i=2}^n (i-1) &\geq \sum_{i=\lfloor n/2 \rfloor + 1}^n n/2 = \lfloor n/2 \rfloor \cdot n/2 = \Omega(n^2) .\end{aligned}$$

2.6.2 Rekurrenzen

In unseren Regeln zur Algorithmenanalyse haben wir uns bisher nicht um Unterprogrammaufrufe gekümmert. Nichtrekursive Unterprogrammaufrufe sind leicht zu handhaben, weil wir das Unterprogramm gesondert analysieren können und die resultierende Schranke in den Ausdruck für die Rechenzeit des aufrufenden Programms einsetzen können. Für rekursive Unterprogramme führt dieser Ansatz aber nicht zu einer geschlossenen Formel, sondern zu einer Rekurrenzrelation.

Beispielsweise haben wir bei der rekursiven Variante der Schulmethode für die Multiplikation als Abschätzung für die Anzahl der Elementaroperationen die Gleichungen $T(1) = 1$ und $T(n) = 4T(\lceil n/2 \rceil) + 4n$ erhalten. Die entsprechenden Gleichungen für den Karatsuba-Algorithmus lauteten $T(n) = 3n^2$ für $n \leq 3$ und $T(n) = 3T(\lceil n/2 \rceil + 1) + 8n$ für $n > 3$. Allgemein definiert eine *Rekurrenzrelation* (kurz: *Rekurrenz*) eine Funktion unter Verwendung von Werten derselben Funktion für kleinere Argumente. Explizite Definitionen für kleine Parameterwerte (Basisfall) vervollständigen die Definition. Das Lösen von Rekurrenzen, d. h. das Auffinden von nichtrekursiven, geschlossenen Formeln für auf diese Weise definierte Funktionen, ist ein interessanter Gegenstand der Mathematik. Hier befassen wir uns hauptsächlich mit den Rekurrenzen, die sich typischerweise bei der Analyse von Teile-und-Herrsche-Algorithmen ergeben. Wir beginnen mit einem einfachen Spezialfall, der die Hauptideen verständlich machen soll. Gegeben ist eine Eingabe der Größe $n = b^k$ für eine natürliche Zahl k . Wenn $k \geq 1$ ist, wenden wir lineare Arbeit auf, um die Instanz in d Teilinstanzen der Größe n/b aufzuteilen und die Ergebnisse der rekursiven Aufrufe auf diesen Teilinstanzen zu kombinieren. Wenn $k = 0$ ist, gibt es keine rekursiven Aufrufe, und wir wenden Arbeit a auf, um die Aufgabe für die Instanz direkt zu lösen.

Satz 2.5 (Mastertheorem (einfache Form)) Für positive Konstanten a , b , c und d , und $n = b^k$ für eine natürliche Zahl k , sei die folgende Rekurrenz gegeben:

$$r(n) = \begin{cases} a & \text{für } n = 1 , \\ d \cdot r(n/b) + cn & \text{für } n > 1 . \end{cases}$$

Dann gilt:

$$r(n) = \begin{cases} \Theta(n) & \text{für } d < b, \\ \Theta(n \log n) & \text{für } d = b, \\ \Theta(n^{\log_b d}) & \text{für } d > b. \end{cases}$$

Abbildung 2.6 illustriert die zentrale Einsicht, die Satz 2.5 zugrundeliegt. Wir betrachten den Zeitaufwand auf jeder Rekursionsebene. Begonnen wird mit einer Instanz der Größe n . Auf der i -ten Rekursionsebene haben wir d^i Teilinstanzen der Größe n/b^i . Die Gesamtgröße aller Instanzen auf Ebene i ist daher

$$d^i \frac{n}{b^i} = n \left(\frac{d}{b} \right)^i.$$

Die Zeit, die (abgesehen von den von ihr verursachten rekursiven Aufrufen) für eine Instanz verbraucht wird, ist c mal die Größe der Instanz, und daher ist der gesamte Zeitaufwand auf einer Rekursionsebene proportional zur Gesamtgröße aller Instanzen auf der Ebene. Je nachdem ob d/b kleiner als 1, gleich 1 oder größer als 1 ist, ergibt sich unterschiedliches Verhalten.

Wenn $d < b$ gilt, dann *verringert sich* der Zeitaufwand *geometrisch* mit der Rekursionsebene und die *erste* Rekursionsebene verursacht schon einen konstanten Anteil der gesamten Ausführungszeit.

Wenn $d = b$ gilt, haben wir auf *jeder* Rekursionsebene *genau den gleichen* Zeitaufwand. Weil es logarithmisch viele Ebenen gibt, ist der gesamte Zeitaufwand $\Theta(n \log n)$.

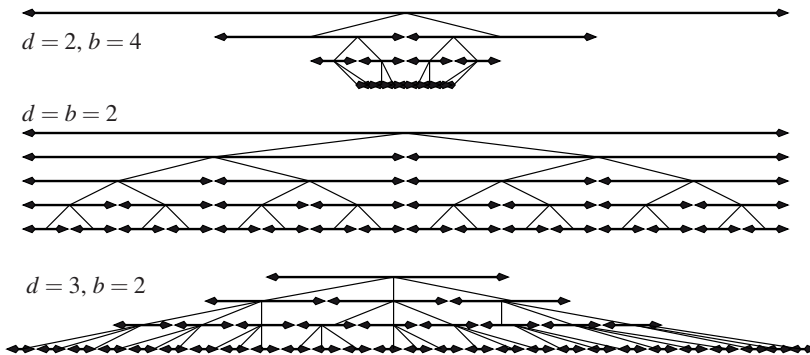


Abb. 2.6. Beispiele für die drei Fälle im Mastertheorem. Instanzen werden durch horizontale Strecken mit Doppelpfeilen angedeutet. Die Länge einer solchen Strecke steht für die Größe der Instanz, und die Teilinstanzen, die aus einer Instanz entstehen, sind in der darunterliegenden Ebene dargestellt. Der obere Teil der Abbildung stellt den Fall $d = 2$ und $b = 4$ dar, bei dem aus einer Instanz zwei Teilinstanzen mit einem Viertel der Größe erzeugt werden – die Gesamtgröße der Teilinstanzen ist nur die Hälfte der Größe der Instanz. Der mittlere Teil der Abbildung stellt den Fall $d = b = 2$ dar, und der untere Teil den Fall $d = 3$ und $b = 2$.

Wenn schließlich $d > b$ gilt, sehen wir einen mit der Rekursionsebene *geometrisch wachsenden* Zeitbedarf, so dass ein konstanter Anteil der Rechenzeit auf der *letzten* Rekursionsebene verbraucht wird. Diese Überlegungen werden nun im Detail ausgeführt.

Beweis. Wir beginnen mit einer Instanz der Größe $n = b^k$. Diese wollen wir Ebene 0 der Rekursion nennen.⁵ Auf Ebene 1 gibt es d Instanzen, jede mit Größe $n/b = b^{k-1}$. Auf Ebene 2 gibt es d^2 Instanzen, jede mit Größe $n/b^2 = b^{k-2}$. Auf Ebene i gibt es d^i Instanzen, jede mit Größe $n/b^i = b^{k-i}$. Auf Ebene k schließlich gibt es d^k Instanzen, jede mit Größe $n/b^k = b^{k-k} = 1$. Jede solche Instanz verursacht Kosten a ; daher sind die gesamten Kosten auf Ebene k genau ad^k .

Als Nächstes berechnen wir die Gesamtkosten der Teile-und-Herrsche-Schritte auf Ebenen 0 bis $k-1$. Auf Ebene i gibt es d^i rekursive Aufrufe für jeweils eine Instanz der Größe b^{k-i} . Jeder solche Aufruf verursacht Kosten $c \cdot b^{k-i}$; die Gesamtkosten auf Level i betragen daher $d^i \cdot c \cdot b^{k-i}$. Damit ergeben sich die Kosten aller Ebenen von 0 bis $k-1$ zusammengenommen zu

$$\sum_{i=0}^{k-1} d^i \cdot c \cdot b^{k-i} = c \cdot b^k \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i.$$

Nun betrachten wir verschiedene Fälle, je nach dem Größenverhältnis von d und b .

Fall $d = b$. Wir haben Kosten $ad^k = ab^k = an = \Theta(n)$ für die letzte Rekursionsebene und Kosten $cnk = cn \log_b n = \Theta(n \log n)$ für die Teile-und-Herrsche-Schritte.

Fall $d < b$. Wir haben Kosten $ad^k < ab^k = an = O(n)$ für die letzte Rekursionsebene. Für die Kosten der Teile-und-Herrsche-Schritte benutzen wir Gleichung (A.13), die die Summe einer geometrischen Reihe angibt, nämlich $\sum_{i=0}^{k-1} q^i = (1 - q^k)/(1 - q)$ für $q > 0$ und $q \neq 1$, und erhalten die Schranken

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1 - (d/b)^k}{1 - d/b} < cn \cdot \frac{1}{1 - d/b} = O(n)$$

und

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1 - (d/b)^k}{1 - d/b} > cn = \Omega(n).$$

Fall $d > b$. Wir rechnen:

$$d^k = (b^{\log_b d})^k = b^{k \log_b d} = (b^k)^{\log_b d} = n^{\log_b d}.$$

⁵ In diesem Beweis verwenden wir die Terminologie von rekursiven Programmen, um Intuition dafür zu wecken, was vor sich geht. Unsere mathematische Überlegung ist aber auf jede Rekurrenz der in Satz 2.5 angegebenen Form anwendbar, auch wenn sie nicht von einem rekursiven Programm herrührt.

Daher hat die letzte Rekursionsebene Kosten $an^{\log_b d} = \Theta(n^{\log_b d})$. Für die Teile- und Herrsche-Schritte benutzen wir wieder die Formel (A.13) für die geometrische Reihe und erhalten

$$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c \frac{d^k - b^k}{d/b - 1} = cd^k \frac{1 - (b/d)^k}{d/b - 1} = \Theta(d^k) = \Theta(n^{\log_b d}). \quad \square$$

Wir werden das Mastertheorem in diesem Buch sehr oft benutzen. Leider wird die Rekurrenz $T(n) \leq 3n^2$ für $n \leq 3$ und $T(n) \leq 3T(\lceil n/2 \rceil + 1) + 8n$ für $n > 3$, die sich beim Algorithmus von Karatsuba ergibt, durch das Mastertheorem in seiner einfachen Form nicht erfasst, da dieses auf- oder abgerundete Formelteile und Ungleichungen nicht vorsieht. Auch kann es vorkommen, dass der additive Term in der Rekurrenz nicht linear ist. Wir zeigen nun, wie man die oberen Schranken aus dem Mastertheorem auf die folgende recht allgemeine *Rekurrenzungleichung* erweitern kann:

$$r(n) \leq \begin{cases} a & , \text{ wenn } n \leq n_0, \\ cn^s + d \cdot r(\lceil n/b \rceil + e_n) & , \text{ wenn } n > n_0. \end{cases} \quad (2.1)$$

Dabei sind $a > 0$, $b > 1$, $c > 0$, $d > 0$ und $s \geq 0$ konstante reelle Zahlen, und die e_n , $n > n_0$, sind ganze Zahlen mit $-\lceil n/b \rceil < e_n \leq e$, für eine ganze Zahl $e \geq 0$.

Satz 2.5 (Mastertheorem (allgemeine Form)) Wenn $r(n)$ die Rekurrenzungleichung (2.1) erfüllt, dann gilt:

$$r(n) = \begin{cases} O(n^s) & \text{für } d < b^s, \text{ d. h. } \log_b d < s, \\ O(n^s \log n) & \text{für } d = b^s, \text{ d. h. } \log_b d = s, \\ O(n^{\log_b d}) & \text{für } d > b^s, \text{ d. h. } \log_b d > s. \end{cases}$$

*Beweis.*⁶ Über die Parameter dürfen wir ohne Beschränkung der Allgemeinheit die folgenden technische Annahmen machen:

- (i) $\lceil n/b \rceil + e < n$ für $n > n_0$,
- (ii) $n_0 \geq 2(e+1)/(1-1/b)$,
- (iii) $a \leq c(n_0+1)^s$.

Wenn nötig, lassen sich (i) und (ii) erreichen, indem man n_0 erhöht. Eventuell muss dann auch a erhöht werden, um sicherzustellen, dass für $n \leq n_0$ die Ungleichung $r(n) \leq a$ gilt. Nachher kann man auch (iii) garantieren, indem man c erhöht.

Wir „glätten“ die Rekurrenz (2.1), indem wir $\hat{r}(n) := \max(\{0\} \cup \{r(i) \mid i \leq n\})$ definieren. Offensichtlich gilt dann $0 \leq r(n) \leq \hat{r}(n)$ für alle n . Daher genügt es, die angegebenen Schranken für $\hat{r}(n)$ zu beweisen. Wir behaupten, dass $\hat{r}(n)$ die folgende einfachere Rekurrenz erfüllt:

$$\hat{r}(n) \leq \begin{cases} a & , \text{ wenn } n \leq n_0, \\ cn^s + d \cdot \hat{r}(\lceil n/b \rceil + e) & , \text{ wenn } n > n_0. \end{cases}$$

⁶ Dieser Beweis kann beim ersten Lesen überblättert werden.

Um dies einzusehen, betrachte ein n . Wenn $n \leq n_0$, gilt offenbar $\hat{r}(n) \leq a$. Sei also $n > n_0$. Dann gibt es ein $i \leq n$ mit $\hat{r}(n) = r(i)$. Wenn $i \leq n_0$, ergibt sich aus Annahme (iii) und $\hat{r}(\lceil n/b \rceil + e) \geq 0$ die Ungleichung $\hat{r}(n) = r(i) \leq a \leq c(n_0 + 1)^s \leq cn^s + d \cdot \hat{r}(\lceil n/b \rceil + e)$. Wenn $i > n_0$, haben wir $\hat{r}(n) = r(i) \leq ci^s + d \cdot r(\lceil i/b \rceil + e_i) \leq cn^s + d \cdot \hat{r}(\lceil n/b \rceil + e)$, weil $\lceil i/b \rceil + e_i \leq \lceil n/b \rceil + e$ gilt.

Nun beweisen wir die in Satz 2.5 behaupteten Schranken für jede Funktion $r(n) \geq 0$, die die Rekurrenz

$$r(n) \leq \begin{cases} a & , \text{ wenn } n \leq n_0, \\ cn + d \cdot r(\lceil n/b \rceil + e) & , \text{ wenn } n > n_0, \end{cases} \quad (2.2)$$

erfüllt, wobei für die Konstanten die Annahmen (i)–(iii) gelten. Daraus folgt der Satz.

Sei $n > n_0$ beliebig. Zunächst betrachten wir die Argumente für $r(\cdot)$, die entstehen, wenn die Rekurrenz (2.2) wiederholt angewendet wird. Sei $N_0 = n$ und $N_i = \lceil N_{i-1}/b \rceil + e$, für $i = 1, 2, \dots$. Nach Annahme (i) gilt $N_i < N_{i-1}$, solange $N_{i-1} > n_0$. Sei k die kleinste Zahl mit $N_k \leq n_0$.

Behauptung 1:

$$N_i \leq 2n/b^i, \text{ für } 0 \leq i < k.$$

Beweis von Beh. 1: Sei $\beta = b^{-1} < 1$. Durch Induktion über i zeigen wir Folgendes:

$$N_i \leq \beta^i n + (e+1) \sum_{0 \leq j < i} \beta^j, \text{ für } 0 \leq i < k. \quad (2.3)$$

Der Fall $i = 0$ ist trivial. Sei also $0 < i < k$. Dann haben wir:

$$\begin{aligned} N_i &= \lceil \beta N_{i-1} \rceil + e \\ &\stackrel{\text{i.V.}}{\leq} \left\lceil \beta \cdot \left(\beta^{i-1} n + (e+1) \sum_{0 \leq j < i-1} \beta^j \right) \right\rceil + e \\ &\leq \beta^i n + (e+1) \sum_{0 \leq j < i-1} \beta^{j+1} + 1 + e \\ &= \beta^i n + (e+1) \sum_{0 \leq j < i} \beta^j. \end{aligned}$$

Weil $\sum_{0 \leq j < i} \beta^j = (1 - \beta^i)/(1 - \beta) < 1/(1 - \beta)$ (siehe (A.13)), folgt aus (2.3) die Ungleichung $n_0 < N_i \leq \beta^i n + (e+1)/(1 - \beta)$. Nach Annahme (ii) haben wir $(e+1)/(1 - \beta) \leq n_0/2$. Es folgt $(e+1)/(1 - \beta) < \beta^i n$, also $N_i \leq 2\beta^i n = 2n/b^i$.

Behauptung 2: $\log_b(n/n_0) \leq k < \log_b(2n/n_0) + 1$ und $b^k = \Theta(n)$.

Beweis von Beh. 2: Weil $2n/b^{k-1} \geq N_{k-1} > n_0$ gilt, haben wir $b^k < 2bn/n_0$, also $b^k = O(n)$, und $k < \log_b(2n/n_0) + 1$. Auf der anderen Seite sieht man mit Induktion ganz leicht, dass $N_i \geq n/b^i$ gilt, für $0 \leq i \leq k$. Daher ist $n_0 \geq N_k \geq n/b^k$; daraus folgt $b^k \geq n/n_0$, also $b^k = \Omega(n)$, und $k \geq \log_b(n/n_0)$.

Wiederholtes Anwenden der Rekurrenz (2.2) liefert Folgendes:

$$\begin{aligned}
r(n) &\leq dr(N_1) + cN_0^s \\
&\leq d^2r(N_2) + cdN_1^s + cN_0^s \\
&\vdots \\
&\leq d^k r(N_k) + cd^{k-1}N_{k-1}^s + \dots + cdN_1^s + cN_0^s \\
&\stackrel{(\text{Beh. 1})}{\leq} d^k a + c \sum_{0 \leq i < k} d^i (2n/b^i)^s \\
&= d^k a + 2^s c \cdot n^s \sum_{0 \leq i < k} (d/b^s)^i.
\end{aligned} \tag{2.4}$$

Fall $d < b^s$. – Dann ist die Summe $\sum_{0 \leq i < k} (d/b^s)^i$ in (2.4) nach (A.13) durch eine Konstante beschränkt, woraus sich $r(n) \leq d^k a + O(n^s)$ ergibt. Weil $d < b^s$, gilt $d^k < (b^k)^s$, und nach Beh. 2 gilt $(b^k)^s = O(n^s)$. Damit erhalten wir $r(n) = O(n^s)$.

Fall $d = b^s$. – Die k Terme in der Summe in (2.4) sind alle gleich 1; daher gilt $r(n) \leq d^k a + 2^s c n^s k$. Mit $d^k = (b^k)^s = O(n^s)$ (nach Beh. 2) erhalten wir $r(n) = O(n^s) + O(n^s k) = O(n^s) + O(n^s \log_b n) = O(n^s \log n)$.

Fall $d > b^s$. – Dann steigen die Terme in der Summe in (2.4) an. Weil $b^k = \Omega(n)$ (nach Beh. 2), also $(b^s)^k = \Omega(n^s)$, liefert (A.13) in diesem Fall

$$n^s \sum_{0 \leq i < k} (d/b^s)^i < n^s \cdot \frac{(d/b^s)^k}{d/b^s - 1} = O(d^k).$$

Mit (2.4) folgt $r(n) = O(d^k)$. Nach Beh. 2 haben wir $d^k = b^{k \log_b d} = O(n^{\log_b d})$, und wir erhalten $r(n) = O(n^{\log_b d})$. \square

Es gibt viele Verallgemeinerungen des Mastertheorems: Man könnte die Rekursion früher abbrechen, die Größe der Teilinstanzen könnten stärker variieren, die Anzahl der Teilinstanzen könnte von der Größe der Instanz abhängen usw. Für weiterführende Informationen verweisen wir den Leser auf die Bücher [90, 190].

Aufgabe 2.8. Betrachten Sie die Rekurrenz

$$C(n) = \begin{cases} 1 & \text{für } n = 1, \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + cn & \text{für } n > 1. \end{cases}$$

Zeigen Sie, dass $C(n) = O(n \log n)$ gilt.

***Aufgabe 2.9.** Gegeben sei ein Teile-und-Herrsche-Algorithmus, dessen Rechenzeit durch die Rekurrenz $T(1) = a$ und $T(n) = \lceil \sqrt{n} \rceil T(\lceil n / \lceil \sqrt{n} \rceil \rceil) + cn$ für $n > 1$ gegeben ist. Zeigen Sie, dass die Rechenzeit $O(n \log \log n)$ ist.

Aufgabe 2.10. Der Aufwand für den Zugriff auf eine Datenstruktur ist oft durch die folgende Rekurrenz gegeben: $T(1) = a$, $T(n) = T(n/2) + c$. Zeigen Sie, dass $T(n) = O(\log n)$ gilt.

2.6.3 „Globale“ Betrachtungen

Die Techniken zur Algorithmenanalyse, die bisher vorgestellt wurden, sind in folgendem Sinn syntaxorientiert: Um ein großes Programm zu analysieren, betrachten wir zunächst seine Teile und fügen die Analyse-Ergebnisse für die Teile zu einer Analyse für das ganze Programm zusammen. Beim Kombinationsschritt werden Summationen und Rekurrenzen benutzt.

Wir werden auch einen völlig anderen Ansatz benutzen, den man „semantikorientiert“ nennen könnte. Dabei ordnen wir Teilen der Programmausführung Teile einer kombinatorischen Struktur zu und argumentieren dann über diese Struktur. Beispielsweise könnten wir feststellen, dass ein bestimmtes Programmsegment für jede Kante eines Graphen höchstens einmal ausgeführt wird und dass daher die Gesamtkosten der Kantenzahl entsprechen. Oder wir könnten feststellen, dass die Ausführung eines bestimmten Programmteils die Größe einer bestimmten Struktur verdoppelt; wenn man zusätzlich weiß, dass die Größe der Struktur am Anfang mindestens 1 ist und bei Beendigung des Programms höchstens n , dann kann die Anzahl der Ausführungen dieses Programmteils höchstens logarithmisch in n sein.

2.7 Analyse des mittleren Falles

In diesem Abschnitt stellen wir die Analyse von Algorithmen im mittleren (oder durchschnittlichen) Fall vor. Hierzu betrachten wir drei Beispiele von zunehmender Komplexität. Wir nehmen dabei an, dass der Leser mit den Grundbegriffen der Wahrscheinlichkeitsrechnung vertraut ist, z. B. mit diskreten Wahrscheinlichkeitsräumen, Erwartungswerten, Indikatorvariablen und der Linearität des Erwartungswerts. In Abschnitt A.3 sind diese Grundlagen kurz dargestellt.

2.7.1 Inkrementieren eines Zählers

Wir beginnen mit einem sehr einfachen Beispiel. Die Eingabe ist ein Array $a[0..n-1]$, in dem Nullen und Einsen stehen. Wir wollen die durch die Bits im Array dargestellte Zahl um 1 erhöhen (modulo 2^n).

```
i := 0
while (i < n and a[i] = 1) do a[i] = 0; i++
if i < n then a[i] = 1
```

Wie oft wird der Rumpf der while-Schleife ausgeführt? Offensichtlich ist n der schlechteste Fall und 0 der beste Fall. Was ist der mittlere Fall? Der erste Schritt bei einer Durchschnittsanalyse ist immer, das Zufallsmodell zu definieren, genauer gesagt, den zugrundeliegenden Wahrscheinlichkeitsraum. Hier legen wir das folgende Zufallsmodell fest: Jede Ziffer hat mit Wahrscheinlichkeit $1/2$ den Wert 0 bzw. 1, und verschiedene Ziffern sind unabhängig. Äquivalent könnte man sagen, dass jede Bitfolge mit n Bits dieselbe Wahrscheinlichkeit hat, nämlich $1/2^n$. Für gegebenes $k \in 0..n$ wird der Schleifenrumpf k -mal ausgeführt genau dann wenn entweder $k < n$

ist und $a[0] = \dots = a[k-1] = 1$ und $a[k] = 0$ gilt oder wenn $k = n$ ist und alle Bits in a den Wert 1 haben. Das erste Ereignis hat Wahrscheinlichkeit $2^{-(k+1)}$, das zweite Wahrscheinlichkeit 2^{-n} . Die mittlere Anzahl von Schleifendurchläufen ist daher

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \leq \sum_{k=0}^{n-1} (k+1)2^{-(k+1)} < \sum_{k \geq 1} k2^{-k} = 2,$$

wobei die letzte Gleichheit im Anhang unter (A.14) zu finden ist.

2.7.2 Zwischenmaxima

Unser zweites Beispiel ist etwas anspruchsvoller. Wir betrachten das folgende einfache Programm, das den größten Eintrag in einem Array $a[1..n]$ findet:

```
 $m := a[1]; \quad \text{for } i := 2 \text{ to } n \text{ do if } a[i] > m \text{ then } m := a[i]$ 
```

Wie oft wird die Zuweisung $m := a[i]$ ausgeführt? Im schlechtesten Fall wird die Zuweisung in jedem Schleifendurchlauf und damit $(n-1)$ -mal ausgeführt, im besten Fall überhaupt nicht. Was ist der mittlere Fall? Wieder beginnen wir mit der Festlegung des Wahrscheinlichkeitsraumes. Wir nehmen an, dass das Array n verschiedene Einträge enthält und dass jede Anordnung dieser Einträge gleich wahrscheinlich ist. Mit anderen Worten: der Wahrscheinlichkeitsraum besteht aus den $n!$ Permutationen der Arrayeinträge. Dabei hat jede Permutation die gleiche Wahrscheinlichkeit, nämlich $1/n!$. Weil es gleichgültig ist, welche konkreten Einträge im Array stehen, nehmen wir an, dass das Array die Zahlen von 1 bis n in irgendeiner Anordnung enthält. Wir fragen nach der mittleren Anzahl der „Zwischenmaxima“. Dabei ist ein Eintrag $a[i]$ ein *Zwischenmaximum*, wenn es größer als alle vorhergehenden Einträge ist. Beispielsweise hat die Folge $\langle 1, 2, 4, 3 \rangle$ drei Zwischenmaxima, die Folge $\langle 3, 1, 2, 4 \rangle$ hat zwei Zwischenmaxima. Die Anzahl der Ausführungen der Zuweisung $m := a[i]$ ist um 1 geringer als die Anzahl der Zwischenmaxima. Für eine Permutation π der Zahlen von 1 bis n sei $M_n(\pi)$ die Anzahl der Zwischenmaxima in π . Was ist $E[M_n]$? Wir werden zwei Überlegungen angeben, mit denen man diesen Erwartungswert ermitteln kann. Für kleine n kann man den Erwartungswert leicht direkt berechnen. Für $n = 1$ gibt es nur eine Permutation, nämlich (1) , mit einem Zwischenmaximum; daher gilt $E[M_1] = 1$. Für $n = 2$ gibt es zwei Permutationen, nämlich $\langle 1, 2 \rangle$ und $\langle 2, 1 \rangle$. Die erste hat zwei Zwischenmaxima, die zweite eines; daher gilt $E[M_2] = 1.5$. Für größere n führt die folgende Überlegung zum Ziel.

Wir schreiben M_n als Summe von Indikatorvariablen I_1 bis I_n , d. h., $M_n = I_1 + \dots + I_n$, wobei $I_k = 1$ für eine Permutation π gilt, wenn der k -te Eintrag in π ein Zwischenmaximum ist. Beispielsweise ist $I_3(\langle 3, 1, 2, 4 \rangle) = 0$ und $I_4(\langle 3, 1, 2, 4 \rangle) = 1$. Es gilt:

$$\begin{aligned} E[M_n] &= E[I_1 + I_2 + \dots + I_n] \\ &= E[I_1] + E[I_2] + \dots + E[I_n] \\ &= \text{prob}(I_1 = 1) + \text{prob}(I_2 = 1) + \dots + \text{prob}(I_n = 1). \end{aligned}$$

Dabei gilt die zweite Gleichung wegen der Linearität des Erwartungswerts (A.2); die dritte Gleichung folgt daraus, dass die I_k 's Indikatorvariablen sind. Es bleibt die Wahrscheinlichkeit dafür zu bestimmen, dass $I_k = 1$ ist. Der k -te Eintrag in einer Zufallspermutation ist ein Zwischenmaximum genau dann wenn der k -te Eintrag der größte der k ersten Einträge ist. In einer Zufallspermutation steht das Maximum der ersten k Einträge mit gleicher Wahrscheinlichkeit in jeder der ersten k Positionen. Daher ist die gesuchte Wahrscheinlichkeit $\text{prob}(I_k = 1) = 1/k$ und es gilt

$$E[M_n] = \sum_{k=1}^n \text{prob}(I_k = 1) = \sum_{k=1}^n \frac{1}{k}.$$

Damit ergibt sich etwa $E[M_4] = 1 + 1/2 + 1/3 + 1/4 = (12 + 6 + 4 + 3)/12 = 25/12$. Die Summe $\sum_{k=1}^n 1/k$ wird in diesem Buch noch häufiger auftauchen. Sie heißt die „ n -te harmonische Zahl“ und wird mit H_n bezeichnet. Man weiß, dass $\ln n \leq H_n \leq 1 + \ln n$ gilt, d. h. $H_n \approx \ln n$, siehe (A.12). Wir erkennen daraus, dass die Zahl von Zwischenmaxima im Mittel viel kleiner ist als im schlechtesten Fall.

Aufgabe 2.11. Zeigen Sie, dass $\sum_{k=1}^n \frac{1}{k} \leq \ln n + 1$ gilt. *Hinweis:* Beweisen Sie, dass

$$\sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx \text{ gilt.}$$

Wir geben jetzt noch eine weitere Methode an, mit der sich die mittlere Anzahl der Zwischenmaxima ermitteln lässt. Dazu setzen wir zur Abkürzung $A_n = E[M_n]$ für $n \geq 1$ und $A_0 = 0$. Der erste Eintrag in einer Permutation ist immer ein Zwischenmaximum, und jede Zahl hat die gleiche Wahrscheinlichkeit $1/n$, an der ersten Stelle zu stehen. Wenn die erste Zahl i ist, können weiter rechts nur noch die Zahlen von $i+1$ bis n Zwischenmaxima sein. Diese $n-i$ Zahlen erscheinen in diesem Rest der Permutation in zufälliger Reihenfolge; daher ist die mittlere Anzahl weiterer Zwischenmaxima genau A_{n-i} . Wir erhalten die folgende Rekurrenzgleichung:

$$A_n = 1 + \frac{1}{n} \cdot \sum_{i=1}^n A_{n-i} \quad \text{oder} \quad nA_n = n + \sum_{i=1}^{n-1} A_i.$$

Diese Gleichung lässt sich mit einem einfachen Trick vereinfachen. Wenn wir die Gleichung für $n-1$ anstelle von n aufschreiben, erhalten wir $(n-1)A_{n-1} = n-1 + \sum_{i=1}^{n-2} A_i$. Wenn wir dies von der Gleichung für n subtrahieren, ergibt sich

$$nA_n - (n-1)A_{n-1} = 1 + A_{n-1} \quad \text{oder} \quad A_n = 1/n + A_{n-1},$$

und damit sofort $A_n = H_n$.

2.7.3 Lineare Suche

Wir kommen jetzt zu unserem dritten Beispiel, das noch etwas mehr Mühe erfordert. Es geht dabei um das folgende Suchproblem. Wir sollen Einträge mit Nummern 1

bis n linear in irgendeiner Reihenfolge anordnen, Eintrag i steht z. B. in Position ℓ_i . Nachdem die Einträge arrangiert worden sind, werden Suchen durchgeführt. Um einen Eintrag mit Schlüssel x zu suchen, durchlaufen wir die Folge von links nach rechts, bis wir auf den Schlüssel x stoßen. Auf diese Weise kostet es ℓ_i Schritte, um auf Eintrag i zuzugreifen.

Es könnte nun sein, dass wir wissen, dass in allen Suchrunden auf die Einträge mit festen Wahrscheinlichkeiten zugegriffen wird; die Wahrscheinlichkeit, dass nach Eintrag Nummer i gesucht wird, wollen wir p_i nennen. Dann ist $p_i \geq 0$ für alle i , $1 \leq i \leq n$, und $\sum_i p_i = 1$. In dieser Situation sind die *erwarteten* oder *mittleren* Kosten einer Suche gleich $\sum_i p_i \ell_i$, weil wir mit Wahrscheinlichkeit p_i nach Eintrag i suchen und die Kosten dieser Suche ℓ_i sind.

Was ist die „beste“ Anordnung für die Einträge, also die Anordnung, die die mittleren Zugriffskosten minimiert? Intuitiv ist es klar, dass wir die Einträge nach fallenden Zugriffswahrscheinlichkeiten ordnen sollten. Dies wollen wir beweisen.

Lemma 2.6. *Eine Anordnung ist optimal bezüglich der mittleren Suchkosten, wenn aus $p_i > p_j$ stets $\ell_i < \ell_j$ folgt. Wenn $p_1 \geq p_2 \geq \dots \geq p_n$, dann führt die Anordnung $\ell_i = i$ für alle i zu optimalen mittleren Suchkosten von $\text{Opt} = \sum_i p_i i$.*

Beweis. Man betrachte eine Anordnung, in der es i und j mit $p_i > p_j$ und $\ell_i > \ell_j$ gibt, d. h., Eintrag i ist wahrscheinlicher als Eintrag j und ist dennoch hinter ihm angeordnet. Wenn wir die Einträge vertauschen, ändern sich die mittleren Suchkosten um den Betrag

$$-(p_i \ell_i + p_j \ell_j) + (p_i \ell_j + p_j \ell_i) = (p_j - p_i)(\ell_i - \ell_j) < 0,$$

d. h., die neue Anordnung ist besser als die alte, und daher kann die erste Anordnung nicht optimal gewesen sein.

Nun betrachten wir den Fall $p_1 > p_2 > \dots > p_n$. Da es nur $n!$ mögliche Anordnungen gibt, existiert eine optimale Anordnung. Wenn $i < j$ ist und wir Eintrag i nach Eintrag j anordnen, dann kann die Anordnung nach der Überlegung im vorigen Absatz nicht optimal sein. Daher setzt die optimale Anordnung Eintrag i an Position $\ell_i = i$, und die resultierende erwartete Suchzeit ist $\sum_i p_i i$.

Wenn $p_1 \geq p_2 \geq \dots \geq p_n$, ist die Anordnung mit $\ell_i = i$ für alle i immer noch optimal. Wenn allerdings einige Wahrscheinlichkeiten gleich sind, gibt es mehr als eine optimale Anordnung. Innerhalb von Blöcken mit gleichen Wahrscheinlichkeiten ist die Anordnung irrelevant. \square

Können wir immer noch etwas Intelligentes tun, wenn wir die Wahrscheinlichkeiten p_i nicht kennen? Die Antwort ist ja, und eine sehr einfache Strategie, die sogenannte „*move-to-front*“-Heuristik“, führt zum Ziel. Sie funktioniert folgendermaßen. Angenommen, in einem Schritt wird Eintrag i gesucht und in Position ℓ_i gefunden. Wenn $\ell_i = 1$, sind wir zufrieden und tun nichts weiter. Andernfalls verschieben wir Eintrag i an Position 1 und verschieben die Einträge an Positionen 1 bis $\ell_i - 1$ um eine Stelle nach hinten. Wir hoffen, dass auf diese Weise Einträge, auf die oft zugegriffen wird, eher näher am Anfang der Anordnung zu finden sind, und dass selten

aufgesuchte Einträge sich eher weiter hinten befinden werden. Wir analysieren jetzt das durchschnittliche Verhalten der *move-to-front*-Heuristik.

Wir nehmen dafür an, dass die n Einträge anfangs in der Liste beliebig angeordnet sind, und betrachten eine feste solche Startanordnung. Als Wahrscheinlichkeitsmodell nehmen wir an, dass eine Folge von Suchrunden stattfindet, wobei in jeder Runde *unabhängig vom bisherigen Verlauf* mit Wahrscheinlichkeit p_i auf Eintrag i zugegriffen wird. Da die Kosten des allerersten Zugriffs auf einen Eintrag i wenig mit dem zufälligen Ablauf zu tun haben, sondern im Wesentlichen von der Startanordnung bestimmt werden, ignorieren wir die Kosten für den ersten Zugriff auf Eintrag i und bewerten ihn pauschal mit Kosten 1.⁷ Wir betrachten eine feste Suchrunde t . Mit C_{MTF} bezeichnen wir die erwarteten Kosten in Runde t . Für jedes i sei ℓ_i die Position von Eintrag i in der Liste zu Beginn von Runde t . Die Größen ℓ_1, \dots, ℓ_n sind Zufallsvariablen, die nur von dem abhängen, was in den ersten $t-1$ Runden passiert ist. Wenn in Runde t auf i zugegriffen wird, sind die Kosten $1 + Z_i$, wo

$$Z_i = \begin{cases} \ell_i - 1 & \text{falls vor } t \text{ schon einmal auf } i \text{ zugegriffen wurde,} \\ 0 & \text{sonst.} \end{cases}$$

Natürlich hängen auch die Zufallsvariablen Z_1, \dots, Z_n nur von dem ab, was in den ersten $t-1$ Runden passiert ist. Daher gilt $C_{MTF} = \sum_i p_i (1 + E[Z_i]) = 1 + \sum_i p_i E[Z_i]$. Nun wollen wir für gegebenes i den Wert $E[Z_i]$ abschätzen. Dazu definieren wir für $j \neq i$ Indikatorzufallsvariablen

$$I_{ij} = \begin{cases} 1 & \text{falls zu Beginn von Runde } t \text{ in der Liste } j \text{ vor } i \text{ steht,} \\ & \text{und vor Runde } t \text{ schon auf mindestens einen der beiden Einträge} \\ & \text{zugegriffen wurde,} \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt $Z_i \leq \sum_{j \text{ mit } j \neq i} I_{ij}$. (Wenn auf i in Runde t erstmals zugegriffen wird, ist $Z_i = 0$. Wenn Runde t nicht den ersten Zugriff auf i darstellt, ist $I_{ij} = 1$ für jedes j , das in der Liste vor i steht, also $Z_i = \sum_{j \text{ mit } j \neq i} I_{ij}$.) Daher gilt auch $E[Z_i] \leq \sum_{j \text{ mit } j \neq i} E[I_{ij}]$. Wir müssen also für jedes $j \neq i$ den Erwartungswert $E[I_{ij}]$ abschätzen.

Wenn es vor Runde t noch keinen Zugriff auf i oder auf j gab, ist $I_{ij} = 0$. Andernfalls betrachten wir die letzte Runde $t_{ij} < t$, in der auf i oder auf j zugegriffen wurde. Die (bedingte) Wahrscheinlichkeit, dass dieser Zugriff j und nicht i galt, dass also $I_{ij} = 1$ gilt, ist $p_j / (p_i + p_j)$. Insgesamt ergibt sich: $E[I_{ij}] = \text{prob}(I_{ij} = 1) \leq p_j / (p_i + p_j)$, also $E[Z_i] \leq \sum_{j \text{ mit } j \neq i} p_j / (p_i + p_j)$.

Durch Summieren über i erhalten wir:

$$C_{MTF} = 1 + \sum_i p_i E[Z_i] \leq 1 + \sum_{i, j \text{ mit } i \neq j} \frac{p_i p_j}{p_i + p_j}.$$

⁷ Insgesamt werden hierdurch höchstens Kosten $n(n-1)$ nicht gezählt. Man kann auch zeigen, dass für eine Suchrunde $t > n$ die durch diese Festlegung verursachte Abweichung von den echten erwarteten Kosten nicht größer als n^2/t ist.

Man beobachtet nun, dass für jedes Paar (i, j) mit $j < i$ der Term $p_i p_j / (p_i + p_j) = p_j p_i / (p_j + p_i)$ doppelt in der Summe erscheint. Um die Notation zu vereinfachen, nehmen wir an, dass $p_1 \geq p_2 \geq \dots \geq p_n$ gilt. Mit $\sum_i p_i = 1$ erhalten wir:

$$\begin{aligned} C_{MTF} &\leq 1 + 2 \sum_{i,j \text{ mit } j < i} \frac{p_i p_j}{p_i + p_j} = \sum_i p_i \left(1 + 2 \sum_{j \text{ mit } j < i} \frac{p_j}{p_i + p_j} \right) \\ &\leq \sum_i p_i \left(1 + 2 \sum_{j \text{ mit } j < i} 1 \right) < \sum_i p_i 2i = 2 \sum_i p_i i = 2 \text{Opt} . \end{aligned}$$

Satz 2.7 *Wenn man die erste Suche nach jedem Eintrag ignoriert, sind die mittleren Suchkosten bei der move-to-front-Heuristik höchstens doppelt so groß wie die mittleren Suchkosten bei optimaler fester Anordnung.*

2.8 Randomisierte Algorithmen

Felizitas darf an einer TV-Spielshow teilnehmen. Das Spiel läuft folgendermaßen ab: Es gibt 100 Kästchen, durchnummeriert von 1 bis 100, die sie in einer beliebigen Reihenfolge öffnen darf. Kästchen i enthält m_i Euro. Felizitas weiß natürlich nicht, was m_i ist, aber sobald sie Kästchen i geöffnet hat, kann sie nachsehen. Verschiedene Kästchen enthalten verschiedene Beträge. Die Spielregeln sind ganz einfach:

- Zu Spielbeginn erhält Felizitas vom Showmaster 10 Spielmarken.
- Wenn sie ein Kästchen öffnet, das einen höheren Betrag als alle vorher geöffneten Kästchen enthält, muss sie eine Spielmarke abgeben.⁸
- Wenn sie eine Marke abgeben müsste, aber keine mehr übrig hat, endet das Spiel und sie verliert alles.
- Wenn es Felizitas gelingt, alle Kästchen zu öffnen und dabei höchstens zehn Marken abzugeben, gewinnt sie und darf alles gefundene Geld behalten.

Die Kästchen sind mit seltsamen Bildern bemalt, und der Showmaster versucht, Felizitas zu helfen, indem er vorschlägt, welches Kästchen sie als Nächstes öffnen sollte. Felizitas' Tante, die keine der Shows verpasst, erzählt ihr, dass nur wenige Kandidaten gewinnen. Felizitas fragt sich nun, ob es sich lohnt, an diesem Spiel teilzunehmen. Gibt es eine Strategie, die ihr eine gute Gewinnchance gibt? Sind die Hinweise des Showmasters nützlich?

Wir wollen zunächst den offensichtlichen Algorithmus analysieren: Felizitas macht immer das, was der Showmaster vorschlägt. Im schlechtesten Fall zeigt er auf Kästchen mit immer steigenden Beträgen. Beim Öffnen eines jeden neuen Kästchens muss sie eine Spielmarke abgeben, und nach dem elften Kästchen hat sie verloren. Die Kandidaten und die Zuschauer würden sich über den Showmaster ärgern,

⁸ Der Betrag im ersten Kästchen ist größer als der in allen bisher geöffneten Kästchen. Das Öffnen des ersten Kästchens kostet Felizitas also auf jeden Fall eine Spielmarke.

die Einschaltquote würde sinken, und er würde gefeuert werden. Die Analyse des schlechtesten Falles hilft uns hier nicht weiter. Der beste Fall wäre, dass der Showmaster sofort das beste Kästchen verrät, also das mit dem größten Betrag. Felizitas wäre glücklich, aber es wäre keine Zeit für die Werbung, und der Showmaster würde auch gefeuert werden. Auch die Analyse des besten Falles gibt uns hier nicht die richtige Information.

Etwas Überlegen führt uns zu der Erkenntnis, dass das Spiel eigentlich nur eine Umformulierung des Zwischenmaxima-Problems aus dem letzten Abschnitt ist. Felizitas muss immer dann eine Spielmarke abgeben, wenn ein neues Maximum auftaucht. Nun wissen wir aus dem letzten Abschnitt, dass die erwartete Anzahl von Zwischenmaxima in einer zufälligen Permutation die n -te harmonische Zahl H_n ist. Für $n = 100$ gilt $H_n < 6$. Würde der Showmaster die Kästchen in zufälliger Anordnung präsentieren, müsste Felizitas im Durchschnitt nur sechs Spielmarken abgeben. Aber weshalb sollte der Showmaster dies tun? Es gibt für ihn keinen Grund, allzu viele Gewinner zu haben.

Die Lösung ist, dass Felizitas ihr Schicksal in die eigene Hand nimmt: *Sie öffnet die Kästchen in zufälliger Reihenfolge*. Sie wählt eines der Kästchen rein zufällig, öffnet es, dann ein weiteres zufällig aus den verbleibenden, öffnet dieses, und immer so weiter. Wie wählt sie ein zufälliges Kästchen? Wenn k viele übrig sind, wählt sie eine Zufallszahl aus dem Bereich von 1 bis k , etwa durch Drehen eines Kreiseis, dessen Rand ein regelmäßiges k -Eck ist. Auf diese Weise erzeugt sie eine zufällige Permutation der Kästchen, und die Analyse aus dem vorherigen Abschnitt kann angewendet werden. Im Mittel wird sie weniger als sechs Spielmarken zurückgeben müssen, und die 10 Spielmarken werden normalerweise ausreichen. – Was wir eben beschrieben haben, ist ein *randomisierter Algorithmus*. Es muss betont werden, dass zwar die mathematische Analyse die gleiche ist wie im Fall, wo der Showmaster die Kästchen in zufälliger Reihenfolge präsentiert, dass aber die Schlussfolgerungen sehr unterschiedlich sind. Im Durchschnitts-Szenario ist man darauf angewiesen, dass der Showmaster tatsächlich eine zufällige Reihenfolge verwendet. Wenn dies der Fall ist, gilt die Analyse; wenn er eine andere Strategie verfolgt, gilt sie nicht. Was wirklich passiert, kann man nicht sagen, höchstens nach vielen Shows und im Nachhinein. Die Situation im Szenario „Randomisierter Algorithmus“ ist völlig anders. Felizitas selbst führt die Zufallsexperimente durch und erzeugt auf diese Weise die Zufallspermutation. Die Analyse gilt immer, gleichgültig was der Showmaster tut.

2.8.1 Das formale Modell

Formal stattdessen wir unsere RAM mit einem zusätzlichen Befehl aus: $R_i := \text{randInt}(R_j)$ weist Register R_i einen zufälligen Wert aus der Menge $0..k-1$ zu, wobei k der Inhalt von R_j ist.

In Pseudocode schreiben wir $v := \text{randInt}(C)$, wobei v eine Integervariable ist. Als Kosten für das Wählen einer solchen Zufallszahl veranschlagen wir eine Zeiteinheit. Algorithmen ohne solche Zufallsexperimente heißen *deterministisch*.

Die Rechenzeit eines randomisierten Algorithmus hängt im Allgemeinen von den zufälligen Entscheidungen ab, die er trifft. Damit ist die Rechenzeit auf einer Eingabe i keine Zahl mehr, sondern eine Zufallsvariable, die von den Ergebnissen der Zufallsexperimente abhängt. Wir könnten die Abhängigkeit der Rechenzeit von den Zufallsexperimenten vermeiden, indem wir unsere Maschine mit einer „Stoppuhr“ ausstatten. Zu Beginn einer Programmausführung stellen wir die Stoppuhr auf einen Wert $T(n)$, der von der Größe n der Eingabe abhängen kann, zählen die Schritte herunter, und halten die Berechnung an, wenn $T(n)$ Schritte ausgeführt worden sind. Wenn die Berechnung auf diese Weise endet, gibt das Programm allerdings keine Antwort zurück.

Auch die Ausgabe eines randomisierten Algorithmus kann von den Ergebnissen der Zufallsexperimente abhängen. Es ist natürlich die Frage, ob ein Algorithmus wirklich nützlich sein kann, dessen Antwort auf einer Eingabe i vom Zufall abhängen kann – wenn die Antwort also heute „Ja“ und morgen „Nein“ sein kann. Wenn die beiden Fälle gleich wahrscheinlich sind, ist die Antwort des Algorithmus natürlich wertlos. Wenn aber die korrekte Antwort viel wahrscheinlicher als die falsche Antwort ist, dann ist die Antwort nützlich. Wir erläutern dies an einem Beispiel.

Alice und Bob sind über eine langsame Telefonleitung miteinander verbunden. Alice hat eine n -Bit-Zahl x , und Bob hat eine n -Bit-Zahl y . Sie wollen herausfinden, ob beide Zahlen gleich sind. Da der Kommunikationskanal langsam ist, möchten sie die Informationsmenge, die ausgetauscht werden muss, möglichst klein halten. Der Aufwand für lokale Berechnungen spielt dagegen keine Rolle.

In der naheliegenden Lösung schickt Alice ihre Zahl an Bob, Bob überprüft, ob die beiden Zahlen gleich sind, und meldet das Ergebnis. Bei diesem Ansatz muss man n Ziffern senden. Alternativ könnte Alice ihre Zahl ziffernweise übermitteln, Bob überprüft Gleichheit für jede Ziffer und meldet das Ergebnis, sobald es feststeht, d. h., sobald zwei Ziffern an derselben Position verschieden sind. Im schlechtesten Fall müssen alle n Ziffern übermittelt werden, nämlich wenn die beiden Zahlen gleich sind. Wir werden nun sehen, dass Randomisierung zu einer dramatischen Verbesserung führt. Nach der Übermittlung von nur $O(\log n)$ Bits kann eine Aussage über Gleichheit oder Ungleichheit gemacht werden, wobei mit großer Wahrscheinlichkeit das richtige Ergebnis herauskommt.

Alice und Bob verfahren nach dem folgenden Protokoll. Jeder von beiden erstellt eine geordnete Liste p_1, \dots, p_L von Primzahlen, und zwar mit den L kleinsten Primzahlen, die größer als 2^k sind. Wir sagen weiter unten, wie k und L (als Funktion von n) zu wählen sind. Auf diese Weise wird sichergestellt, dass Alice und Bob die gleiche Liste erzeugen. Dann wählt Alice zufällig einen Index i mit $1 \leq i \leq L$ und sendet i und $x_A \bmod p_i$ an Bob. Dieser berechnet $x_B \bmod p_i$. Wenn $x_A \bmod p_i \neq x_B \bmod p_i$ gilt, meldet er, dass die Zahlen unterschiedlich sind, sonst, dass sie gleich sind. Wenn $x_A = x_B$ gilt, wird Bob offensichtlich immer „gleich“ sagen. Wenn jedoch $x_A \neq x_B$ ist und trotzdem $x_A \bmod p_i = x_B \bmod p_i$ gilt, dann wird Bob die beiden Zahlen fälschlicherweise für gleich erklären. Wie groß ist die Wahrscheinlichkeit für einen solchen Irrtum?

Ein Fehler passiert, wenn $x_A \neq x_B$, aber $x_A \equiv x_B \pmod{p_i}$ gilt. Die letztere Bedingung ist äquivalent dazu, dass p_i die Differenz $D = x_A - x_B$ teilt. Der Absolutbe-

trag dieser Differenz kann nicht größer als 2^n sein. Weil jedes p_i größer als 2^k ist, enthält die Liste von Alice und Bob höchstens n/k Primzahlen, die D teilen.⁹ Daher ist die Fehlerwahrscheinlichkeit nicht größer als $(n/k)/L$. Diese Wahrscheinlichkeit können wir beliebig klein machen, indem wir L genügend groß wählen. Wenn wir etwa die Wahrscheinlichkeit kleiner als $0.000001 = 10^{-6}$ machen wollen, wählen wir $L = 10^6(n/k)$.

Wie sollte man aber k wählen? Für genügend große k sind von den Zahlen in $[2^k \dots 2^{k+1} - 1]$ etwa $2^k / \ln(2^k) \approx 1.4427 \cdot 2^k / k$ viele Primzahlen.¹⁰ Wenn also $2^k/k \geq 10^6 n/k$ gilt, wird die Liste nur Zahlen mit $k+1$ Bits enthalten. Die Bedingung $2^k \geq 10^6 n$ ist gleichbedeutend mit $k \geq \log n + 6 \log 10$. Diese Wahl von k führt dazu, dass Alice $\log L + k + 1 = \log n + 12 \log 10 + 1$ Bits sendet. *Dies ist eine exponentielle Verbesserung gegenüber dem naiven Protokoll!*

Wie wäre eine Fehlerwahrscheinlichkeit von weniger als 10^{-12} zu erreichen? Wir könnten analog rechnen wie eben, mit $L = 10^{12}n/k$. Alternativ könnten wir das Protokoll zweimal laufen lassen, und die Zahlen genau dann als gleich erklären, wenn beide Durchläufe sie für gleich erklären. Dieses 2-Stufen-Protokoll macht nur dann einen Fehler, wenn beide Stufen einen Fehler machen; daher ist die Fehlerwahrscheinlichkeit höchstens $10^{-6} \cdot 10^{-6} = 10^{-12}$.

Aufgabe 2.12. Vergleichen Sie die Effizienz der beiden Methoden, mit denen man eine Fehlerwahrscheinlichkeit von 10^{-12} erreicht.

Aufgabe 2.13. Im oben beschriebenen Protokoll müssen Alice und Bob unrealistisch lange Listen von Primzahlen bereitstellen. Diskutieren Sie das folgende, veränderte Protokoll: Alice wählt eine zufällige Zahl p mit $k+1$ Bits (und führender 1) und testet p darauf, ob es sich um eine Primzahl handelt. Wenn dies nicht der Fall ist, wiederholt sie den Versuch. Wenn p eine Primzahl ist, sendet sie p und $x_A \bmod p$ an Bob. Bob berechnet $x_B \bmod p$ und vergleicht.

Aufgabe 2.14. Nehmen Sie an, Sie haben einen randomisierten Algorithmus, der mit Wahrscheinlichkeit höchstens $1/4$ das falsche Ergebnis liefert. Sie lassen den Algorithmus k -mal laufen und geben das Mehrheitsergebnis aus, das häufiger als $k/2$ -mal erscheint (falls es existiert). Leiten Sie eine Schranke für die Fehlerwahrscheinlichkeit her, als Funktion von k . Für $k=2$ und $k=3$ sollten Sie exakt rechnen, für größere k eine Schranke angeben. Beschreiben Sie dann, wie k zu wählen ist, um eine Fehlerwahrscheinlichkeit zu erreichen, die kleiner als eine gegebene Zahl ε ist.

⁹ Sei d die Anzahl der Primzahlen in der Liste, die D teilen. Dann gilt $2^n \geq |D| \geq (2^k)^d = 2^{kd}$, also $d \leq n/k$.

¹⁰ Für $x \geq 1$ bezeichne $\pi(x)$ die Anzahl der Primzahlen, die kleiner oder gleich x sind. Beispielsweise ist $\pi(10) = 4$, weil es vier Primzahlen (2, 3, 5 und 7) gibt, die kleiner oder gleich 10 sind. Dann gelten für alle $x \geq 55$ die Ungleichungen $x/(\ln x + 2) < \pi(x) < x/(\ln x - 4)$. Weitere Informationen findet man im (englischen) Wikipedia-Eintrag zu „Prime Number Theorem“.

2.8.2 Las-Vegas- und Monte-Carlo-Algorithmen

Randomisierte Algorithmen gibt es im Wesentlichen in zwei Varianten, nämlich Las-Vegas-Algorithmen und Monte-Carlo-Algorithmen. Ein *Las-Vegas-Algorithmus* berechnet immer das richtige Ergebnis, aber seine Rechenzeit auf einer Eingabe i ist eine Zufallsvariable. Unsere Lösung für die Spielshow entspricht einem Las-Vegas-Algorithmus, wenn wir ihn immer weiter nach dem Kästchen mit dem maximalen Wert suchen lassen; allerdings ist die Anzahl der Zwischenmaxima, also die Anzahl der abzuliefernden Spielmarken, eine Zufallsvariable. Ein Monte-Carlo-Algorithmus hat immer die gleiche Rechenzeit, aber es gibt eine positive Wahrscheinlichkeit dafür, dass er ein falsches Ergebnis liefert. Die Wahrscheinlichkeit, dass das Ergebnis falsch ist, ist höchstens $1/4$. Unser Algorithmus, der zwei Zahlen über eine Telefonverbindung vergleicht, ist ein Monte-Carlo-Algorithmus. In Aufgabe 2.14 wird gezeigt, dass bei Monte-Carlo-Algorithmen (durch Wiederholen) die Fehlerwahrscheinlichkeit beliebig klein gemacht werden kann.

Aufgabe 2.15. Nehmen Sie an, Sie haben einen Las-Vegas-Algorithmus mit erwarteter Rechenzeit $t(n)$. Sie lassen ihn $4t(n)$ Schritte lang laufen. Wenn er innerhalb dieser erlaubten Zeit ein Ergebnis liefert, wird dieses als Antwort zurückgegeben, andernfalls wird eine beliebige Ausgabe zurückgegeben. Zeigen Sie, dass der resultierende Algorithmus ein Monte-Carlo-Algorithmus ist.

Aufgabe 2.16. Nehmen sie an, Sie haben einen Monte-Carlo-Algorithmus mit Rechenzeit $m(n)$, der mit Wahrscheinlichkeit mindestens p eine korrekte Antwort liefert, und einen deterministischen Algorithmus, der in Zeit $v(n)$ überprüfen kann, ob das vom Monte-Carlo-Algorithmus gelieferte Ergebnis korrekt ist. Beschreiben Sie, wie aus diesen beiden Algorithmen ein Las-Vegas-Algorithmus mit erwarteter Rechenzeit $(m(n) + v(n))/(1 - p)$ gewonnen werden kann.

Zum Schluss kommen wir nochmals auf das Spielshow-Beispiel zurück. Felizitas hat 10 Spielmarken, die erwartete Anzahl von verbrauchten Spielmarken ist kleiner als 6. Wie sicher kann sie sein, als Gewinnerin nach Hause zu gehen? Wir müssen dazu die Wahrscheinlichkeit dafür abschätzen, dass M_n größer als 11 ist, denn sie verliert genau dann, wenn die Anzahl der Zwischenmaxima in der Folge der gefundenen Geldbeträge 11 oder größer ist. Die *Markov-Ungleichung* erlaubt eine solche Abschätzung. Sie besagt, dass für eine beliebige nichtnegative Zufallsvariable X und eine beliebige Konstante $c \geq 1$ die Ungleichung $\text{prob}(X \geq c \cdot \mathbb{E}[X]) \leq 1/c$ gilt, s. (A.4). Dies wenden wir für $X = M_n$ und $c = 11/6$ an. Wir erhalten

$$\text{prob}(M_n \geq 11) \leq \text{prob}\left(M_n \geq \frac{11}{6} \mathbb{E}[M_n]\right) \leq \frac{6}{11},$$

und daher ist die Gewinnwahrscheinlichkeit größer als $5/11$.

2.9 Graphen

Graphen sind ein für die Algorithmik äußerst nützliches Konzept. Wir benutzen sie immer, wenn wir Objekte und Beziehungen zwischen ihnen zu modellieren haben;

in der Terminologie der Graphen heißen die Objekte *Knoten* und die Beziehungen zwischen Paaren von Knoten heißen *Kanten*. Ziemlich offensichtliche Anwendungen von Graphen sind Straßenkarten und Kommunikationsnetzwerke, aber es gibt auch abstraktere Anwendungen. Beispielsweise könnten die Knoten Teilaufgaben darstellen, die beim Bau eines Hauses durchgeführt werden müssen, wie etwa „Bauen der Mauern“ oder „Einsetzen der Fenster“, und Kanten könnten Präzedenzbeziehungen (Vorrangsbeziehungen) sein wie „die Mauern müssen gebaut werden, bevor die Fenster eingesetzt werden können“. Wir werden auch viele Datenstrukturen kennenlernen, bei denen auf ganz natürliche Weise jedes Objekt als ein Knoten aufzufassen ist und jeder Zeiger als gerichtete Kante von dem Objekt, von er ausgeht, zu dem Objekt, auf das er verweist.

Wenn Menschen Überlegungen mit Graphen anstellen, finden sie es normalerweise bequem, mit Bildern zu arbeiten, die Knoten als kleine Kreise und Kanten als gerade oder gekrümmte Linien oder als Pfeile darstellen. Zur algorithmischen Verarbeitung von Graphen benötigen wir eine mehr mathematisch orientierte Notation: ein *gerichteter Graph* $G = (V, E)$ ist ein Paar, bestehend aus einer *Knotenmenge* (oder *Eckenmenge*) V und einer Kantenmenge (oder *Bogenmenge*) $E \subseteq V \times V$. Manchmal schreiben wir kurz „*Digraph*“ für „gerichteter Graph“ (engl.: *directed graph*). Zum Beispiel ist in Abb. 2.7 der gerichtete Graph $G = (\{s, t, u, v, w, x, y, z\}, \{(s, t), (t, u), (u, v), (v, w), (w, x), (x, y), (y, z), (z, s), (s, v), (z, w), (y, t), (x, u)\})$ bildlich dargestellt. Im ganzen Buch soll gelten, dass $n = |V|$ und $m = |E|$ ist, wenn n und m keine anderen Bedeutungen zugeordnet sind. Eine Kante $e = (u, v) \in E$ stellt eine Verbindung von u nach v dar. Wir nennen u den *Startknoten* und v den *Zielknoten* von e . Wir sagen, dass e *inzident* mit u und v ist, oder dass u und v *auf* e *liegen*, und dass u und v *adjacent* sind. Oft heißt auch u ein (unmittelbarer) *Vorgänger* von v und v ein (unmittelbarer) *Nachfolger* von u . Der Spezialfall einer (*Eigen*-)*Schleife* (v, v) ist verboten, außer er wird ausnahmsweise ausdrücklich erlaubt.

Der *Ausgangsgrad* eines Knotens v ist die Zahl der Kanten, die von v ausgehen, sein *Eingangsgrad* ist die Zahl der Kanten, die in v enden, formal $\text{outdegree}(v) = |\{u \in V \mid (v, u) \in E\}|$ und $\text{indegree}(v) = |\{u \in V \mid (u, v) \in E\}|$. Zum Beispiel hat Knoten w in Graph G in Abb. 2.7 Eingangsgrad 2 und Ausgangsgrad 1.

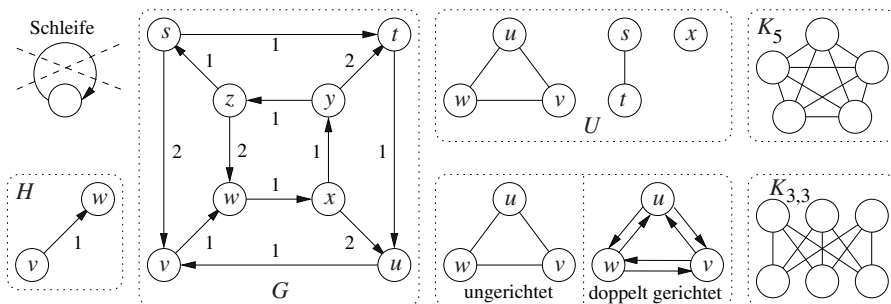


Abb. 2.7. Einige Graphen

Ein *doppelt gerichteter Graph* ist ein Digraph, in dem für jede gerichtete Kante (u, v) auch die Gegenkante (v, u) vorhanden ist. Ein *ungerichteter Graph* lässt sich als verschlankte Darstellung eines doppelt gerichteten Graphen auffassen, in der wir das Kantenpaar $(u, v), (v, u)$ als die Paarmenge $\{u, v\}$ schreiben. Abb. 2.7 zeigt einen ungerichteten Graphen mit drei Knoten und sein doppelt gerichtetes Gegenstück. Die meisten graphentheoretischen Begriffe werden für ungerichtete Graphen genauso definiert wie für Digraphen; daher werden wir uns in diesem Abschnitt auf die Digraphen konzentrieren und ungerichtete Graphen nur erwähnen, wenn es Abweichungen gibt. Zum Beispiel hat ein ungerichteter Graph nur halb so viele Kanten wie sein doppelt gerichtetes Gegenstück. Wenn $\{u, v\}$ eine Kante ist, heißt u auch ein *Nachbar* von v ; in der doppelt gerichteten Version ist u sowohl Vorgänger als auch Nachfolger von v . Der Eingangsgrad und der Ausgangsgrad eines Knotens in einem ungerichteten Graphen ist immer gleich, so dass wir hier einfach vom *Grad* eines Knotens sprechen. Ungerichtete Graphen sind deshalb wichtig, weil Richtungen häufig keine Rolle spielen und weil viele Probleme in ungerichteten Graphen leichter zu lösen sind als in allgemeinen Digraphen; manche Probleme sind überhaupt nur in ungerichteten Graphen sinnvoll.

Ein Graph $G' = (V', E')$ ist ein *Teilgraph* des Graphen $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. Wenn $G = (V, E)$ und eine Teilmenge $V' \subseteq V$ gegeben sind, dann ist der durch V' *induzierte* Teilgraph definiert durch $G' = (V', E \cap (V' \times V'))$. In Abb. 2.7 wird durch die Knotenmenge $\{v, w\}$ in G der Teilgraph $H = (\{v, w\}, \{(v, w)\})$ induziert. Eine Teilmenge $E' \subseteq E$ der Kanten induziert den Teilgraphen (V, E') .

Oft werden Knoten oder Kanten mit zusätzlichen Informationen verknüpft. Insbesondere werden wir oft *Kantengewichte* oder *Kantenkosten* $c: E \rightarrow \mathbb{R}$ benötigen, die jeder Kante einen Zahlenwert zuordnen. Beispielsweise hat die Kante (z, w) im Graphen G in Abb. 2.7 das Gewicht $c((z, w)) = -2$. Man beachte, dass in einem ungerichteten Graphen eine Kante $\{u, v\}$ nur ein Gewicht haben kann, wohingegen in doppelt gerichteten Graphen auch $c((u, v)) \neq c((v, u))$ gelten kann.

Die letzte Seite enthielt viele trockene Definitionen in gedrängter Form. Wenn der Leser sie „in Aktion“ sehen möchte, findet er in Kap. 8 Algorithmen, die auf Graphen arbeiten. Aber auch an dieser Stelle wird das Material noch etwas interessanter.

Ein wichtiger etwas „höherer“ graphentheoretischer Begriff ist der eines Weges oder Pfades. Ein *Weg* (oder *Pfad*) $p = \langle v_0, \dots, v_k \rangle$ in einem gerichteten Graphen $G = (V, E)$ ist eine Knotenfolge mit der Eigenschaft, dass aufeinanderfolgende Knoten durch Kanten in E verbunden sind, d. h., dass $(v_0, v_1) \in E, (v_1, v_2) \in E, \dots, (v_{k-1}, v_k) \in E$ gilt. Die Zahl $k \geq 0$ heißt die *Länge* von p ; der Weg *verläuft* (oder *führt*) von v_0 nach v_k . Manchmal wird ein Weg auch durch die Folge seiner Kanten dargestellt. Beispielsweise ist im Digraphen G in Abb. 2.7 $\langle u, v, w \rangle = \langle (u, v), (v, w) \rangle$ ein Weg der Länge 2. In einem ungerichteten Graphen ist $p = \langle v_0, \dots, v_k \rangle$ ein Weg, wenn p Weg in dem entsprechenden doppelt gerichteten Graphen ist und für $1 \leq i < k$ stets $v_{i-1} \neq v_{i+1}$ gilt. (Es ist also nicht erlaubt, dass dieselbe Kante in einer Richtung und unmittelbar darauf in der anderen Richtung benutzt wird.) Im Graphen U in Abb. 2.7 ist $\langle u, w, v, u, w, v \rangle$ ein Weg der Länge 5. Ein Weg $\langle v_0, \dots, v_k \rangle$ heißt *einfach*, wenn seine Knoten, außer eventuell v_0 und v_k , paarweise verschieden sind. Im Di-

graphen G in Abb. 2.7 ist $\langle z, w, x, u, v, w, x, y \rangle$ ein Weg, der nicht einfach ist. Es ist leicht einzusehen, dass Folgendes gilt: Wenn es einen Weg von u nach v gibt, dann gibt es auch einen einfachen Weg von u nach v .

Kreise (oder *Zyklen*) sind Wege der Länge mindestens 1, bei denen der erste und der letzte Knoten zusammenfallen. In ungerichteten Graphen verlangt man, dass die erste und die letzte Kante des Weges verschieden sind, woraus sich ergibt, dass Kreise Länge mindestens 3 haben. (In G in Abb. 2.7 ist $\langle u, v, w, x, y, z, w, x, u \rangle$ ein Kreis, in U ist $\langle u, w, v, u, w, v, u \rangle$ ein Kreis.) Ein *einfacher* Kreis ist ein einfacher Weg, der ein Kreis ist. Folgendes ist leicht einzusehen: Wenn G einen Kreis hat, dann auch einen einfachen Kreis. Ein einfacher Kreis, der alle Knoten eines Graphen besucht, heißt *Hamiltonkreis*. (In Abb. 2.7 sind $\langle s, t, u, v, w, x, y, z, s \rangle$ in G und $\langle w, u, v, w \rangle$ in U Hamiltonkreise.)

Die Begriffe „Weg“ und „Kreis“ erlauben es uns, noch komplexere Begriffe zu definieren. Ein Digraph heißt *stark zusammenhängend*, wenn es von jedem Knoten u zu jedem anderen Knoten v einen Weg gibt. Graph G in Abb. 2.7 ist stark zusammenhängend. Eine *starke Zusammenhangskomponente* von G ist ein maximaler knoteninduzierter stark zusammenhängender Teilgraph von G . Wenn wir in Abb. 2.7 aus G die Kante (w, x) entfernen, erhalten wir einen Digraphen, der überhaupt keine Kreise enthält. Ein Digraph ohne Kreise heißt *azyklisch* oder *kreisfrei*. Azyklische Digraphen heißen kurz *DAG* (engl.: *directed acyclic graph*). In einem DAG besteht jede starke Zusammenhangskomponente aus genau einem Knoten. Ein ungerichteter Graph heißt *zusammenhängend*, wenn es von jedem Knoten u aus einen (ungerichteten) Weg zu jedem anderen Knoten v gibt. Die Zusammenhangskomponenten sind die maximalen zusammenhängenden Teilgraphen. Innerhalb einer Zusammenhangskomponente sind jeweils zwei Knoten durch Wege verbunden; von einer Zusammenhangskomponente in eine andere führt kein Weg. Zum Beispiel hat Graph U in Abb. 2.7 die Zusammenhangskomponenten $\{u, v, w\}$, $\{s, t\}$ und $\{x\}$. Die Knotenmenge $\{u, w\}$ induziert einen zusammenhängenden Teilgraphen, aber dieser ist nicht maximal und bildet daher keine Zusammenhangskomponente.

Aufgabe 2.17. Beschreiben Sie zehn grundlegend verschiedene Anwendungen, die unter Verwendung von Graphen modelliert werden können. (Straßen- und Radwegnetze sind *nicht* grundlegend verschieden!) Mindestens fünf dieser Anwendungen sollten in diesem Buch nicht vorkommen.

Aufgabe 2.18. Ein Graph heißt *planar*, wenn er auf einem Blatt Papier so gezeichnet werden kann, dass sich nirgends zwei Kanten überkreuzen. Erklären Sie, weshalb Straßennetzwerke *nicht* unbedingt planar sein müssen. Zeigen Sie, dass die Graphen K_5 und $K_{3,3}$ in Abb. 2.7 nicht planar sind.

2.9.1 Ein erster Graphalgorithmus

Es ist Zeit für einen Beispiellgorithmus. Wir beschreiben einen Algorithmus, der testet, ob ein gerichteter Graph azyklisch ist oder nicht. Der Ansatz ist die einfache Beobachtung, dass ein Knoten v mit Ausgangsgrad 0 nicht auf einem Kreis liegen

kann. Wenn wir also v (und seine Eingangskanten) streichen, erhalten wir einen Graphen G' , der genau dann azyklisch ist, wenn G azyklisch ist. Diese Transformation wird wiederholt ausgeführt, bis einer von zwei Fällen eintritt: Entweder erhalten wir den leeren Graphen ohne jeden Knoten, der sicherlich azyklisch ist, oder wir erhalten einen Graphen G^* , in dem jeder Knoten Ausgangsgrad mindestens 1 hat. Im letzteren Fall findet man leicht einen Kreis: Man beginnt an einem beliebigen Knoten und wählt in jedem Knoten, den man eben erreicht hat, eine beliebige Ausgangskante, bis ein Knoten v' erreicht wird, den man schon vorher einmal gesehen hat. Der so konstruierte Weg hat die Form $(v, \dots, v', \dots, v')$, und das Schlussteil (v', \dots, v') bildet einen Kreis. In Abb. 2.7 beispielsweise hat der Graph G keinen Knoten mit Ausgangsgrad 0. Um einen Kreis zu finden, könnten wir bei Knoten z starten und dem Weg $\langle z, w, x, u, v, w \rangle$ folgen, bis wir zum zweitenmal auf den Knoten w treffen. Damit haben wir dann den Kreis $\langle w, x, u, v, w \rangle$ gefunden. Anders verläuft die Ausführung des Algorithmus, wenn wir aus G die Kante (w, x) entfernen. Dann gibt es keinen Kreis, und unser Algorithmus streicht alle Knoten in der Reihenfolge w, v, u, t, s, z, y, x . In Kap. 8 werden wir sehen, wie man Graphen in einer Weise darstellen kann, dass der hier skizzierte Algorithmus in linearer Zeit, also in Zeit $O(|V| + |E|)$, abläuft. (Siehe hierzu auch Aufgabe 8.3.) Der Algorithmus kann leicht so ergänzt werden, dass er zertifizierend ist: Wenn der Algorithmus einen Kreis findet, ist der Graph sicherlich zyklisch, und es kann leicht getestet werden, ob die ausgegebene Knotenfolge ein Kreis in G ist. Wenn der Algorithmus alle Knoten aus G entfernt, nummerieren wir die Knoten in der Reihenfolge durch, in der sie aus G gestrichen werden. Wenn wir in einem Schritt einen Knoten v mit Ausgangsgrad 0 entfernen, müssen in G alle Kanten, die v als Startknoten haben, zu früher entfernten Knoten gehen, also zu solchen mit einer niedrigeren Nummer als v . Die Nummerierung beweist also, dass G kreisfrei ist: Entlang jeder Kante sind die Knotennummern fallend, was sich wiederum leicht verifizieren lässt.

Aufgabe 2.19. Finden Sie für beliebiges n einen DAG mit n Knoten, der $n(n-1)/2$ Kanten besitzt. Zeigen Sie, dass kein DAG mit n Knoten mehr als $n(n-1)/2$ Kanten haben kann.

2.9.2 Bäume

Ein ungerichteter Graph heißt ein *Baum*, wenn es zwischen zwei beliebigen Knoten jeweils *genau* einen Weg gibt; ein Beispiel ist in Abb. 2.8 dargestellt. Ein ungerichteter Graph heißt ein *Wald*, wenn es zwischen zwei beliebigen Knoten *höchstens* einen Weg gibt. Man sieht sofort, dass jede Zusammenhangskomponente eines Waldes ein Baum ist.

Lemma 2.8. Für einen ungerichteten Graphen G sind die folgenden Aussagen äquivalent:

- (1) G ist ein Baum.
- (2) G ist zusammenhängend und hat genau $n-1$ Kanten.
- (3) G ist zusammenhängend und besitzt keine Kreise.

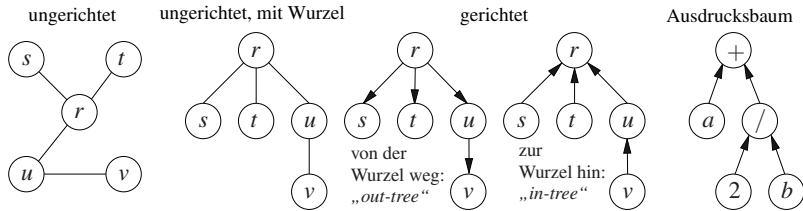


Abb. 2.8. Verschiedene Arten von Bäumen. Von *links* nach *rechts* sehen wir einen ungerichteten Baum, einen ungerichteten Baum mit Wurzel, einen von der Wurzel weg und einen zur Wurzel hin gerichteten Baum und einen arithmetischen Ausdruck.

Beweis. In einem Baum gibt es zwischen je zwei Knoten einen eindeutig bestimmten Weg. Daher ist ein Baum zusammenhängend und enthält keinen Kreis. Umgekehrt gilt: Wenn es in G zwei Knoten gibt, die durch zwei unterschiedliche Wege verbunden sind, dann enthält G einen Kreis. (Man betrachte zwei Knoten u und v und zwei verschiedene Wege von u nach v , wobei die Summe der Längen dieser Wege über alle möglichen solchen Situationen minimal ist. Wegen der Minimalität müssen die beiden Wege u auf unterschiedlichen Kanten verlassen, und sie können sich erst in Knoten v wieder treffen. Daher bilden die beiden Wege zusammen einen einfachen Kreis.) Damit haben wir die Äquivalenz von (1) und (3) bewiesen. Wir zeigen nun, dass auch (2) und (3) äquivalent sind. Dazu betrachten wir einen zusammenhängenden Graphen $G = (V, E)$; wir setzen $m = |E|$. Nun führen wir in Gedanken das folgende Experiment durch: Wir beginnen mit dem („leeren“) Graphen mit Knotenmenge V und keiner Kante und fügen die Kanten aus E eine nach der anderen hinzu. Das Hinzufügen einer Kante verringert die Anzahl der Zusammenhangskomponenten um höchstens 1. Da wir mit $n = |V|$ Komponenten beginnen und am Ende nur noch eine Komponente vorhanden ist, muss die Anzahl m der Kanten mindestens $n - 1$ sein. Angenommen nun, das Hinzufügen der Kante $e = \{u, v\}$ würde die Anzahl der Komponenten nicht um 1 verringern. Dann sind u und v bereits durch einen Weg verbunden, und das Hinzufügen von e erzeugt einen Kreis. Wenn G kreisfrei ist, kann dies also nicht eintreten, und wir erhalten $m = n - 1$. Daher folgt (2) aus (3). Nun nehmen wir an, dass G zusammenhängend ist und genau $n - 1$ Kanten hat. Wieder fügen wir die Kanten nacheinander in einen anfangs leeren Graphen ein. Wenn nun durch die Kante $e = \{u, v\}$ ein Kreis geschlossen wird, sind u und v schon verbunden und das Einfügen von e verringert die Anzahl der Komponenten nicht. Damit bleibt am Ende mehr als eine Zusammenhangskomponente übrig, ein Widerspruch. Damit folgt auch (3) aus (2). \square

Lemma 2.8 gilt nicht für Digraphen. Insbesondere kann ein DAG viel mehr als $n - 1$ Kanten haben. Ein gerichteter Graph heißt ein *Out-Tree* (von der Wurzel weg gerichteter Baum), mit Wurzel r , wenn es von r aus zu jedem Knoten genau einen Weg gibt. Er heißt ein *In-Tree* (zur Wurzel hin gerichteter Baum) mit Wurzel r , wenn es von jedem Knoten aus genau einen Weg nach r gibt. Abbildung 2.8 zeigt einige Beispiele. Die *Tiefe* eines Knotens v in einem Baum mit Wurzel ist die Länge des

Function $eval(r) : \mathbb{R}$

```

if  $r$  ist ein Blatt then return die in  $r$  gespeicherte Zahl
else                                                                 //  $r$  ist ein Operatorknoten
     $v_1 := eval(\text{erstes Kind von } r)$ 
     $v_2 := eval(\text{zweites Kind von } r)$ 
    return  $v_1 \operatorname{operator}(r) v_2$  // wende den in  $r$  gespeicherten Operator an

```

Abb. 2.9. Rekursive Auswertung eines Ausdrucksbaums mit Wurzel r .

Weges zwischen v und der Wurzel. Die *Höhe* (oder auch *Tiefe*) eines Baums mit Wurzel ist die maximale Tiefe eines Knotens.

Man kann auch einen ungerichteten Baum mit einer Wurzel versehen, indem man einfach einen beliebigen Knoten zur Wurzel erklärt. Informatiker haben die merkwürdige Angewohnheit, Bäume so zu zeichnen, dass die Wurzel ganz oben sitzt und alle Kanten von oben nach unten verlaufen. Für Bäume mit Wurzeln ist es üblich, Beziehungen zwischen Knoten mit Wörtern zu bezeichnen, die aus der Welt der Verwandtschaftsbeziehungen stammen. Kanten verlaufen zwischen einem eindeutigen *Vorgängerknoten* (engl.: *parent*¹¹) und seinen *Kindknoten* (engl.: *child*). Knoten mit demselben Vorgängerknoten heißen *Geschwisterknoten* (engl.: *sibling*). Knoten ohne Kinder heißen *Blätter* (engl.: *leaf*). Knoten, die keine Blätter sind, heißen *innere* Knoten. Wenn u auf dem Weg von der Wurzel zu einem Knoten v liegt und verschieden von v ist, nennen wir u einen Vorfahren von v und entsprechend v einen Nachkommen von u . Ein Knoten u und alle seine Nachkommen bilden einen *Teilbaum* mit Wurzel u . Wir betrachten ein Beispiel: In den gerichteten Bäumen in Abb. 2.8 ist r die Wurzel; s , t , und v sind Blätter; s , t und u sind Geschwisterknoten, weil sie Kindknoten desselben Vorgängerknotens r sind; u ist ein innerer Knoten; r und u sind die Vorfahren von v ; s , t , u und v sind die Nachfahren von r ; v und u bilden einen Teilbaum mit Wurzel u .

2.9.3 Geordnete Bäume

Bäume eignen sich ausgezeichnet zur Darstellung von Hierarchien. Man betrachte beispielsweise den arithmetischen Ausdruck $a + 2/b$. Wir wissen, dass dieser Ausdruck bedeuten soll, dass a und $2/b$ addiert werden sollen. Es ist aber gar nicht so einfach, dies aus der Zeichenfolge $\langle a, +, 2, /, b \rangle$ abzulesen. Es muss zum Beispiel die Punkt-vor-Strich-Regel beachtet werden. Compiler isolieren dieses syntaktische Wissen in *Parsern*, die aus dem Formeltext eine strukturiertere, baumbasierte Darstellung erzeugen. Aus unserem Beispielausdruck würde der Ausdrucksbaum ganz rechts in Abb. 2.8 entstehen. Solche Bäume sind gerichtet und im Gegensatz zu graphentheoretischen Bäumen *geordnet*, d. h., die Kinder eines jeden inneren Knotens sind angeordnet. In unserem Beispiel ist der a -Knoten das erste Kind der Wurzel, der $/$ -Knoten das zweite.

¹¹ Anm. d. Ü.: Leider gibt es im Deutschen kein genau passendes Wort im Singular.

Ausdrucksbäume lassen sich leicht durch einen einfachen rekursiven Algorithmus auswerten. Abbildung 2.9 gibt einen Algorithmus zur Auswertung von Ausdrucksbäumen an, deren Blätter Zahlen enthalten und deren innere Knoten Operatoren enthalten (etwa $+$, $-$, \cdot , $/$).

In diesem Buch werden uns viele weitere Beispiele für geordnete Bäume begegnen. In Kapiteln 6 und 7 werden sie für die Darstellung von grundlegenden Datenstrukturen benutzt, in Kap. 12 für die systematische Erkundung von Lösungsräumen.

2.10 P und NP

Wann sollten wir einen Algorithmus „effizient“ nennen? Gibt es Probleme, für die kein effizienter Algorithmus existiert? Natürlich ist es immer etwas willkürlich, wo man die Grenze zwischen „effizienten“ und „ineffizienten“ Algorithmen ziehen möchte. Jedoch hat sich die folgende Unterscheidung als nützlich erwiesen: Ein Algorithmus \mathcal{A} läuft in *polynomieller Zeit* oder ist ein *Polynomialzeitalgorithmus*, wenn es ein Polynom $p(n)$ gibt, so dass die Ausführungszeit von \mathcal{A} auf Eingaben der Größe n in $O(p(n))$ liegt. Wenn nichts anderes gesagt wird, soll die Größe einer Eingabe immer in Bits gemessen werden. Ein Problem *kann in polynomieller Zeit gelöst werden*, wenn es einen Polynomialzeitalgorithmus dafür gibt. Wir setzen nun „effizient lösbar“ gleich mit „lösbar in polynomieller Zeit“. Ein großer Vorteil dieser Definition ist, dass Implementierungsdetails normalerweise keine Rolle spielen. Beispielsweise ist es irrelevant, ob eine clevere Datenstruktur die Rechenzeit eines $O(n^3)$ -Algorithmus um den Faktor n verringern kann. In allen Kapiteln dieses Buches, außer in Kap. 12, geht es um effiziente Algorithmen.

Es gibt viele wichtige Probleme, für die es zwar im Prinzip Algorithmen gibt, für die man aber keine *effizienten* Algorithmen kennt. Hier wollen wir nur sechs Beispiele erwähnen:

- Das Hamiltonkreisproblem: Gegeben ist ein ungerichteter Graph. Entscheide, ob er einen Hamiltonkreis enthält.
- Das Erfüllbarkeitsproblem für Boolesche Formeln: Gegeben ist eine Boolesche Formel in *konjunktiver Normalform*. Entscheide, ob sie eine erfüllende Belegung besitzt. – Dabei ist eine Boolesche Formel in konjunktiver Normalform eine Konjunktion $C_1 \wedge C_2 \wedge \dots \wedge C_k$ von *Klauseln*. Eine Klausel ist eine Disjunktion $(\ell_1 \vee \ell_2 \vee \dots \vee \ell_h)$ von *Literalen*, und ein Literal ist eine Boolesche Variable x_i oder eine negierte Boolesche Variable $\neg x_i$. Zum Beispiel ist $(x_1 \vee \neg x_3 \vee \neg x_9)$ eine Klausel.
- Das Cliquesproblem: Gegeben ist ein ungerichteter Graph und eine natürliche Zahl k . Entscheide, ob der Graph einen vollständigen Teilgraphen (eine *Clique*) mit k Knoten enthält. – Dabei heißt ein Graph *vollständig*, wenn jedes Paar von Knoten durch eine Kante verbunden ist; ein Beispiel ist der Graph K_5 in Abb. 2.7.
- Das Rucksackproblem: Gegeben sind n Paare (w_i, p_i) von natürlichen Zahlen sowie natürliche Zahlen M und P . Entscheide, ob es eine Teilmenge $I \subseteq 1..n$ gibt, für die $\sum_{i \in I} w_i \leq M$ und $\sum_{i \in I} p_i \geq P$ gilt. – Anschaulich ist dies die Frage,

ob man bei n gegebenen Objekten, wobei Objekt i Volumen w_i und Profit p_i hat, eine Auswahl der Objekte in einen Rucksack mit Volumen M packen kann, so dass der gesamte Profit mindestens P ist.

- Das Problem des Handlungsreisenden (TSP): Gegeben ist ein kantengewichteter ungerichteter Graph und eine natürliche Zahl C . Entscheide, ob der Graph einen Hamiltonkreis der Länge höchstens C besitzt. (Siehe Abschnitt 11.6.2 für Details.)
- Das Graphfärbungsproblem: Gegeben ist ein ungerichteter Graph und eine natürliche Zahl $k > 0$. Entscheide, ob es eine Färbung der Knoten des Graphen mit k Farben gibt, so dass adjazente Knoten immer verschieden gefärbt sind.

Die Tatsache, dass wir für diese Probleme keine effizienten Algorithmen kennen, beweist natürlich nicht, dass es solche Algorithmen nicht gibt. Man weiß einfach nicht, ob es solche Algorithmen gibt oder nicht. Insbesondere kennt man auch keinen Beweis dafür, dass es für diese Probleme keine effizienten Algorithmen geben kann. Im allgemeinen ist es sehr schwierig zu beweisen, dass ein gegebenes Problem nicht innerhalb einer bestimmten Zeitschranke gelöst werden kann. (Einige einfache untere Schranken werden wir in Kap. 5.3 kennenlernen.) Die meisten Algorithmiker glauben, dass es für die sechs oben aufgeführten Probleme keine effizienten Algorithmen gibt.

Die Komplexitätstheorie hat einen interessanten Ersatz für die fehlenden Beweise für untere Schranken gefunden. Sie fasst algorithmische Probleme in große Gruppen („Komplexitätsklassen“) zusammen, so dass die Probleme in einer Gruppe im Bezug auf ein Komplexitätsmaß äquivalent sind. Insbesondere gibt es eine große Klasse von äquivalenten Problemen, die man **NP-vollständig** nennt. Dabei ist „NP“ eine Abkürzung für „nichtdeterministische polynomielle Zeit“. Für das Verständnis des Folgenden ist es unerheblich, ob der Leser den Begriff „nichtdeterministische polynomielle Zeit“ kennt oder nicht. Die sechs oben genannten Probleme sind **NP-vollständig**, so wie viele andere natürliche Probleme. Im Rest dieses Abschnittes werden wir die Klasse **NP** und die Klasse der **NP-vollständigen** Probleme formal definieren. Für eine vollständige Darstellung der Theorie verweisen wir den Leser auf Bücher über Berechenbarkeitstheorie und Komplexitätstheorie [16, 79, 197, 219].

Wie in der Komplexitätstheorie üblich, nehmen wir an, dass Eingaben als Zeichenreihen über einem festen endlichen Alphabet Σ mit $|\Sigma| \geq 2$ kodiert sind. (Man denke an das ASCII- oder Unicode-Alphabet oder die Binärokodierungen dieser Alphabete. Im letzteren Fall ist $\Sigma = \{0, 1\}$.) Die Menge aller Zeichenreihen (oder Wörter) mit Buchstaben aus Σ heißt Σ^* . Für $x = a_1 \dots a_n \in \Sigma^*$ wird die Anzahl $|x| = n$ der Zeichen in x als Größenmaß benutzt. Ein *Entscheidungsproblem* ist eine Teilmenge $L \subseteq \Sigma^*$. Mit $\chi_L: \Sigma^* \rightarrow \{0, 1\}$ (gelesen „chi“) bezeichnen wir die *charakteristische Funktion* von L , d. h., $\chi_L(x) = 1$, falls $x \in L$, und $\chi_L(x) = 0$, falls $x \notin L$. Ein Entscheidungsproblem ist *in polynomieller Zeit lösbar*, wenn seine charakteristische Funktion in polynomieller Zeit berechenbar ist. Mit **P** bezeichnen wir die Klasse der Entscheidungsprobleme, die in polynomieller Zeit lösbar sind. Ein Entscheidungsproblem L liegt in **NP**, wenn es ein Prädikat $Q(x, y)$, (d. h. eine Menge $Q \subseteq (\Sigma^*)^2$) und ein Polynom p gibt, so dass Folgendes gilt:

- (1) Für jedes $x \in \Sigma^*$ gilt $x \in L$ genau dann wenn es ein $y \in \Sigma^*$ mit $|y| \leq p(|x|)$ und $Q(x, y)$ gibt;
- (2) die charakteristische Funktion von Q ist in polynomieller Zeit berechenbar.

Wenn $x \in L$ und $Q(x, y) = \text{true}$ gilt, nennen wir y einen *Zeugen*, *Beweis* oder *Beleg für x* (d. h. für die Tatsache, dass $x \in L$ ist). Für unsere Beispielprobleme kann man leicht zeigen, dass sie in **NP** liegen. Beim Hamiltonkreisproblem ist der Zeuge ein Hamiltonkreis im Eingabegraphen. Ein Zeuge für eine Boolesche Formel ist eine Belegung für die Variablen, die die Formel wahr macht. Die Lösbarkeit einer Instanz des Rucksackproblems wird durch eine Teilmenge der Objekte belegt, die vom Volumen her in den Rucksack passen und die Profitschranke erreichen.

Aufgabe 2.20. Beweisen Sie, dass das Cliquesproblem, das Handlungsreisendenproblem (TSP) und das Graphfärbungsproblem in **NP** liegen.

Es wird weithin vermutet, dass **P** eine echte Teilmenge von **NP** ist. Obgleich es gute Argumente für diese Vermutung gibt, wie wir gleich sehen werden, ist sie bisher nicht bewiesen. Sie würde insbesondere nach sich ziehen, dass es für **NP**-vollständige Probleme keine effizienten Algorithmen gibt.

Ein Entscheidungsproblem ist *L* *polynomialzeitreduzierbar* (oder einfach *reduzierbar*) auf ein Entscheidungsproblem L' , wenn es eine in polynomieller Zeit berechenbare Funktion g gibt, so dass für alle $x \in \Sigma^*$ gilt: $x \in L$ genau dann wenn $g(x) \in L'$. Wenn L auf L' reduzierbar ist und $L' \in \mathbf{P}$ gilt, dann folgt recht leicht $L \in \mathbf{P}$. (Sei ein Algorithmus für die Reduktionsfunktion g mit polynomieller Zeitschranke $p(n)$ und ein Algorithmus für $\chi_{L'}$ mit polynomieller Zeitschranke $q(n)$ gegeben. Ein Algorithmus für χ_L arbeitet wie folgt: Zu Eingabe x berechne $g(x)$ in Zeit $\leq p(|x|)$, dann teste, ob $g(x) \in L'$ ist. Weil Turingmaschinen pro Schritt nur ein Zeichen schreiben können, gilt $|g(x)| \leq |x| + p(|x|)$, daher kostet dieser Test Zeit höchstens $q(|x| + p(|x|))$. Dies ist ebenfalls polynomiell in $|x|$.) Ähnlich zeigt man, dass die Reduzierbarkeit eine transitive Relation ist. Ein Entscheidungsproblem L heißt **NP-schwer** (engl.: **NP-hard**), wenn *jedes* Problem in **NP** auf L polynomialzeitreduzierbar ist. Ein Problem heißt **NP-vollständig**, wenn es **NP-schwer** ist und in **NP** liegt. Auf den ersten Blick scheint es sehr schwierig zu sein, zu beweisen, dass irgendein Problem **NP-vollständig** ist – schließlich muss man beweisen, dass *jedes* Problem in **NP** darauf reduzierbar ist. Jedoch gelang genau dies im Jahr 1971: Cook und Levin bewiesen (unabhängig voneinander), dass das Erfüllbarkeitsproblem für Boolesche Formeln **NP-vollständig** ist [49, 132]. Von da ab war es „leicht“. Nehmen wir an, es soll bewiesen werden, dass L **NP-vollständig** ist. Dafür muss man zwei Dinge zeigen: (1) $L \in \mathbf{NP}$, und (2) es gibt ein schon bekanntes **NP-vollständiges** Problem L' , das auf L reduzierbar ist. Mit jedem neuen **NP-vollständigen** Problem wird es leichter, zu beweisen, dass andere Probleme **NP-vollständig** sind. Auf der Webseite <http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html> findet man einen Katalog von **NP-vollständigen** Problemen, der laufend aktualisiert wird. Wir geben ein Beispiel für eine Reduktion an.

Lemma 2.9. *Das Erfüllbarkeitsproblem für Boolesche Formeln ist polynomialzeitreduzierbar auf das Cliquesproblem.*

Beweis. Sei $F = C_1 \wedge \dots \wedge C_k$ eine Boolesche Formel in konjunktiver Normalform, mit $C_i = (\ell_{i1} \vee \dots \vee \ell_{ih_i})$ und $\ell_{ij} = x_{ij}^{\beta_{ij}}$. Dabei ist x_{ij} eine Boolesche Variable, und $\beta_{ij} \in \{0, 1\}$. Ein hochgestellter Index 0 zeigt dabei eine negierte Variable an. Man betrachte den folgenden Graphen G . Seine Knoten stellen die Literale in unserer Formel dar, d. h. $V = \{r_{ij} : 1 \leq i \leq k \text{ und } 1 \leq j \leq h_i\}$. Zwei Knoten r_{ij} und $r_{i'j'}$ sind genau dann durch eine Kante verbunden, wenn $i \neq i'$ und entweder $x_{ij} \neq x_{i'j'}$ oder $\beta_{ij} = \beta_{i'j'}$ gilt. In Worten: Die Knoten zu zwei Literalen sind verbunden, wenn diese Literale zu verschiedenen Klauseln gehören und es eine Belegung gibt, die beide gleichzeitig erfüllt. Wir behaupten, dass F genau dann erfüllbar ist, wenn G eine Clique der Größe k hat.

Wir nehmen zuerst an, dass es eine Belegung α gibt, die F erfüllt. Diese Belegung muss mindestens ein Literal in jeder Klausel wahr machen, beispielsweise Literal $\ell_{i,j(i)}$ in Klausel C_i , für $1 \leq i \leq k$. Betrachte nun den Teilgraphen von G , der von den Knoten $r_{i,j(i)}$, $1 \leq i \leq k$, aufgespannt wird. Dies ist eine Clique mit k Knoten. Wären nämlich $r_{i,j(i)}$ und $r_{i',j(i')}$ nicht durch eine Kante verbunden, dann müsste $x_{i,j(i)} = x_{i',j(i')}$ und $\beta_{i,j(i)} \neq \beta_{i',j(i')}$ gelten. Dann wären aber die Literale $\ell_{i,j(i)}$ und $\ell_{i',j(i')}$ Komplemente voneinander, und α könnte nicht beide erfüllen.

Nehmen wir nun umgekehrt an, dass G eine Clique K der Größe k besitzt. Wir können dann eine erfüllende Belegung α für F konstruieren. Für jedes i , $1 \leq i \leq k$, enthält K genau einen Knoten, den wir $r_{i,j(i)}$ nennen. Wir konstruieren eine erfüllende Belegung für F , indem wir $\alpha(x_{i,j(i)}) = \beta_{i,j(i)}$ setzen. Die hiervon nicht erfassten Variablen werden beliebig belegt. Man beachte, dass α wohldefiniert ist, weil $x_{i,j(i)} = x_{i',j(i')}$ impliziert, dass $\beta_{i,j(i)} = \beta_{i',j(i')}$ ist; sonst wären die Knoten $r_{i,j(i)}$ und $r_{i',j(i')}$ nicht durch eine Kante verbunden. Es ist klar, dass α die Formel F erfüllt. \square

Aufgabe 2.21. Zeigen Sie, dass das Hamiltonkreisproblem auf das Handlungsreisendenproblem (TSP) polynomialzeitreduzierbar ist.

Aufgabe 2.22. Zeigen Sie, dass das Graphfärbungsproblem auf das Erfüllbarkeitsproblem für Boolesche Formeln polynomialzeitreduzierbar ist.

Alle **NP**-vollständigen Probleme sitzen sozusagen „in einem Boot“. Falls es jemandem gelingen sollte, auch nur für *eines* von ihnen einen Polynomialzeitalgorithmus zu finden, dann folgt **NP** = **P**. Weil sehr viele Leute schon vergeblich versucht haben, solche Algorithmen zu finden, wird es immer unwahrscheinlicher, dass dies jemals passieren wird. Die **NP**-vollständigen Probleme sind wechselseitig Belege dafür, dass sie schwer zu lösen sind.

Kann man die Theorie der **NP**-Vollständigkeit auch auf Optimierungsprobleme anwenden? Optimierungsprobleme (siehe Kap. 12) lassen sich leicht in Entscheidungsprobleme umwandeln. Anstatt nach einer optimalen Lösung zu suchen, fragen wir einfach, ob es eine zulässige Lösung mit einem Zielfunktionswert mindestens k gibt, wobei k eine zusätzliche Eingabe ist. Auch die Umkehrung gilt: Wenn wir einen Algorithmus haben, der entscheidet, ob es eine zulässige Lösung mit Zielfunktionswert k oder größer gibt, können wir mittels einer Kombination von exponentieller und binärer Suche (siehe Abschnitt 2.5) den optimalen Zielfunktionswert finden.

Ein Algorithmus für ein Entscheidungsproblem gibt die Antwort „ja“ oder „nein“, je nachdem ob die Eingabe zur entsprechenden Sprache gehört oder nicht. Er gibt aber keinen Zeugen an. Häufig kann man aber Zeugen konstruieren, indem man den Entscheidungsalgorithmus mehrfach (auch auf veränderte Versionen der Eingabe) anwendet. Nehmen wir an, wir wissen, dass ein Graph G eine Clique der Größe k enthält, und wollen eine solche finden, haben aber nur einen Algorithmus, der entscheidet, ob ein Graph eine Clique einer bestimmten Größe enthält. Wenn $k = 1$ ist, bildet ein beliebiger Knoten in G die gesuchte Clique. Sonst wählen wir in G einen beliebigen Knoten v und fragen, ob $G' = G \setminus v$ (dies steht für den Graphen G ohne v und die mit v inzidenten Kanten) eine Clique der Größe k enthält. Falls ja, suchen wir in G' rekursiv nach einer Clique dieser Größe. Falls nein, muss v in G zu jeder Clique der Größe k gehören. In diesem Fall betrachten wir die Menge V' der Nachbarn von v und suchen rekursiv in dem von V' induzierten Teilgraphen eine Clique der Größe $k - 1$. Dann ist $\{v\} \cup V'$ eine Clique der Größe k in G .

2.11 Implementierungsaspekte

Unser Pseudocode lässt sich leicht in Programme in einer beliebigen imperativen Programmiersprache übertragen. Für C++ and Java stellen wir unten etwas genauere Hinweise bereit. Die Programmiersprache Eiffel [150] ermöglicht das Arbeiten mit Zusicherungen, Invarianten, Vor- und Nachbedingungen.

Unsere speziellen Werte \perp , $-\infty$ und ∞ werden für Gleitkommazahlen von den Programmiersprachen bereitgestellt. Für andere Datentypen müssen diese Werte emuliert werden. Beispielsweise könnte man die kleinste und die größte darstellbare ganze Zahl als $-\infty$ bzw. ∞ benutzen. Undefinierte Zeiger werden oft durch einen Nullzeiger **null** dargestellt. Manchmal benutzen wir die speziellen Werte \perp , $-\infty$ und ∞ nur aus Bequemlichkeit; eine robuste Implementierung sollte ihre Verwendung vermeiden. In späteren Kapiteln werden wir Beispiele hierfür sehen.

Randomisierte Algorithmen benötigen Zugang zu einer Quelle für Zufälligkeit. Dabei hat man die Wahl zwischen einem Hardware-Zufallsgenerator, der echte Zufallszahlen erzeugt, und einem algorithmischen Generator, der Pseudo-Zufallszahlen erzeugt. Für mehr Information hierzu verweisen wir den Leser auf die (englische) Wikipedia-Seite zu „*random number generation*“.

2.11.1 C++

Unseren Pseudocode kann man als eine kompakte Schreibweise für eine Teilmenge von C++ ansehen. Die Speicherverwaltungsoperationen **allocate** und **dispose** ähneln den Operationen *new* und *delete* in C++. In C++ wird bei der Erzeugung eines Arrays für jeden Arrayeintrag der Standardkonstruktor für den Typ des Eintrags aufgerufen, d. h., die Bereitstellung eines Arrays der Länge n kostet Zeit $\Omega(n)$, wohingegen diese Operation bei Arrays mit n *ints* konstante Zeit benötigt. In unserem Ansatz findet diese Initialisierung nicht statt, stattdessen nehmen wir an, dass Arrays, die nicht explizit initialisiert wurden, beliebige Inhalte („garbage“) enthalten.

In C++ kann man diesen Effekt durch Benutzung der C-Funktionen *malloc* und *free* erzielen. Von diesem veralteten Vorgehen ist allerdings dringend abzuraten; es sollte nur in Ausnahmefällen benutzt werden, in denen die Initialisierung von Arrays zu einer zu starken Verschlechterung der Rechenzeit führen würde. Wenn die Speicherverwaltung für viele kleine Objekte rechenzeitkritisch ist, kann man sie mittels der *allocator*-Klasse der C++Standardbibliothek an die jeweilige Situation anpassen.

Unsere Parametrisierung von Klassen mittels **of** ist ein Spezialfall des Template-Mechanismus von C++. Die Parameter, die bei einer Objektdекlaration dem Klassennamen in runden Klammern angefügt werden, entsprechen in C++ den Parametern eines Konstruktors.

Zusicherungen werden als C-Makros im include-File `assert.h` implementiert. Im Standardfall lösen verletzte Zusicherungen einen Laufzeitfehler aus; die Fehlermeldung enthält die Position der Zusicherung im Programmtext. Durch Definieren des Makros *NDEBUG* lässt sich diese Überprüfung von Zusicherungen abschalten.

Für viele der Datenstrukturen und Algorithmen, die in diesem Buch diskutiert werden, gibt es in verschiedenen Softwarebibliotheken exzellente Implementierungen. Gute Quellen sind die Standard Template Library STL [171], die BOOST-Bibliotheken [29] für C++ und die LEDA-Bibliothek [143, 130] von effizienten Datenstrukturen und Algorithmen.

2.11.2 Java

Java besitzt keine explizite Speicherverwaltung. Vielmehr werden durch ein Speicherbereinigungsprogramm (engl.: *garbage collector*) in regelmäßigen Abständen Speichersegmente, die nicht mehr benutzt werden, identifiziert und für die Wiederverwendung bereitgestellt. Während dieser Ansatz die Programmierung sehr vereinfacht, kann er sich sehr negativ auf den Zeitaufwand auswirken. Methoden, die dem abhelfen, gehören nicht zum Themenbereich dieses Buches. Sogenannte generische Typen machen es möglich, Klassen zu parametrisieren. Zusicherungen werden mit der *assert*-Anweisung implementiert.

Hervorragende Implementierungen für viele Datenstrukturen und Algorithmen findet man im Paket *java.util* und in der Datenstrukturenbibliothek JDSL [87].

2.12 Historische Anmerkungen und weitere Ergebnisse

Die Definition des das RAM-Modells für die Algorithmenanalyse wurde von Shepherdson und Sturgis [195] vorgeschlagen. Dieses Modell erlaubt es nur, Zahlen mit einer logarithmischen Anzahl von Bits in einer Zelle zu speichern. Wenn man diese Einschränkung fallen lässt, ergeben sich unerwünschte Konsequenzen; beispielsweise fallen die Komplexitätsklassen **P** und **PSPACE** zusammen [97]. Ein detaillierteres abstraktes Maschinenmodell wurde von Knuth [124] beschrieben.

Floyd [68] führte das Konzept der Invarianten ein, um Programmen eine Bedeutung zuzuordnen, und Hoare [101, 102] wendete sie systematisch an. Das Buch [90]

ist ein Kompendium über den Umgang mit Summen und Rekurrenzen und vielen anderen Themen der „zählenden“ Kombinatorik, die für die Algorithmenanalyse hilfreich sind.

In Büchern über Compilerbau (z. B. [156, 221]) findet man weitere Informationen über die Übersetzung von in höheren Programmiersprachen geschriebenen Programmen in Maschinencode.

Algorithmen und Datenstrukturen

Die Grundwerkzeuge

Dietzfelbinger, M.; Mehlhorn, K.; Sanders, P.

2014, XII, 380 S. 101 Abb., Softcover

ISBN: 978-3-642-05471-6