Skript zur Vorlesung

Algorithmen und Datenstrukturen

Herbstsemester 2016

Stand: 15. Dezember 2017

Markus Püschel
Peter Widmayer
Daniel Graf
Tobias Pröger

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Liesst man in der Zeitung Artikel über Algorithmen, dann bekommt man oft den Eindruck, Algorithmen seien ein eher böses Ding, welchem einem empfiehlt rote Kugelschreiber zu kaufen, weil man kürzlich blaue Kugelschreiber gekauft hat. Wir als Informatiker verstehen darunter aber etwas anderes, nämlich eine systematische Anleitung zum Lösung von Problemen wie etwa die Multiplikation zweier Zahlen. Viele Algorithmen kennt man bereits aus dem alltäglichen Leben, ohne sie explizit als Algorithmus zu erkennen: So sind etwa schon Kochrezepte oder IKEA-Anleitungen Beispiele für Algorithmen.

Algorithmen im Alltag

Als einführendes Beispiel betrachten wir folgende Situation: Eine Kuh steht auf einer Weide, entlang derer ein Zaun verläuft. Die Weide ist komplett abgegrast, aber auf der anderen Seite des Zauns gibt es noch viel Gras. Glücklicherweise hat der Zaun ein Loch, durch das die Kuh auf die gegenüberliegende Seite des Zauns wechseln kann. Die Kuh weiss aber nicht, wo das Loch ist. Wir nehmen nun vereinfachend an, die Kuh sei punktförmig, und der Zaun sei eine unendliche lange Gerade. Die Kuh kann jedes Mal einen Schritt nach links oder nach rechts machen, und sie erkennt das Loch im Zaun nur dann, wenn sie direkt davor steht. Wir nehmen an, dass der initiale Abstand zwischen Kuh und Loch exakt D Schritte betrage. Abbildung 1.1 zeigt die Situation für $D = 4$.

Kuhproblem

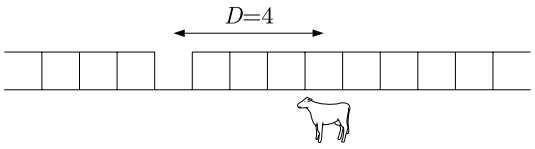


Abb. 1.1 Situation beim Kuhproblem mit $D = 4$.

Wie sollte die Kuh nun vorgehen, um zum Loch zu kommen? Offensichtlich ist es nicht ausreichend, einfach in irgendeine Richtung (z.B. nach rechts) zu laufen, bis das Loch gefunden wird, denn der Zaun ist unendlich lang und das Loch könnte in der entgegengesetzten Richtung (also links von der Kuh) liegen. Das Problem liegt also darin, dass die Kuh nicht weiss, in welche Richtung sie laufen muss. Ein möglicher Ausweg besteht darin, hin- und her zu laufen, also zunächst einen Schritt nach rechts zu machen, dann zwei Schritte nach links, dann drei Schritte nach rechts, usw. Auf diese Weise wird die Kuh das Loch auf jeden Fall nach endlicher Zeit finden. Wie viele Schritte werden dazu benötigt? Offensichtlich macht die Kuh $1 + 2 + \cdots + (2D - 1)$ viele Schritte, bis sie zum ersten Mal D Schritte weit vom Ausgangspunkt

Lösung

Anzahl Schritte

entfernt ist. Sie befindet sich dann rechts vom Ausgangspunkt. Liegt das Loch aber links vom Ausgangspunkt, dann benötigt die Kuh weitere $2D$ Schritte, um zum Loch zu kommen. Insgesamt findet sie also mit diesem Verfahren nach spätestens $1 + 2 + \cdots + 2D$ vielen Schritten das Loch im Zaun. Tatsächlich werden wir in der Übung sehen, dass bessere Strategien gibt, die das Loch (zumindest für grosse D) mit weniger Schritten finden.

Algorithmen und ihre Bedeutung

Um die Rolle von Algorithmen in der Informatik einzuschätzen, werfen wir einen Blick auf eine Definition der Informatik, die 1986 von der US-amerikanischen Informatikervereinigung ACM spezifiziert wurde. Diese besagt:

> "Computer science is the systematic study of algorithms and data structures, specifically
>
> 1) their formal properties,
> 2) their mechanical and linguistic realizations, and
> 3) their applications."

Algorithmen sind also *das* zentrale Thema der Informatik (siehe auch die Folien zur ersten Vorlesung). In dieser Vorlesung werden wir viele Algorithmen kennenlernen und ihre formalen Eigenschaften wie Korrektheit oder Effizienz studieren. Man könnte einwenden, dass bei den heutigen Rechenleistungen die Effizienz von Algorithmen nur noch eine untergeordnete Rolle spielt. Das folgende Beispiel verdeutlicht, dass diese Aussage naiv ist.

Korrektheit Effizienz

Problem des Handlungsreisenden

Dazu betrachten wir das *Problem des Handlungsreisenden* (engl.: *Travelling Salesman Problem*, TSP). Es seien n Städte auf einer Landkarte gegeben, und wir wollen annehmen, dass zwischen jedem Paar von zwei Städten i und j eine Strassenverbindung der Länge d_{ij} existiert. Eine *Rundreise* startet in einer Stadt s, besucht danach jede andere Stadt $s' \in \{1, \ldots, n\} \backslash \{s\}$ genau einmal und führt schlussendlich zum Startpunkt s zurück. Gesucht wird nun eine Rundreise mit minimaler Gesamtlänge.

Wenn wir die Reihenfolge festlegen, in der wir die Städte besuchen möchten, dann kann die gesamte Reiselänge mit $n - 1$ Additionen ermittelt werden, indem die Abstände zwischen den entsprechenden Städten aufsummiert werden. Für eine *feste* Reise kann ihre Länge also effizient ermittelt werden. Leider gibt $(n-1)!$ viele mögliche Rundreisen, was enorm viel ist. Zur Veranschaulichung: Schon für $n = 43$ Städte existieren mehr mögliche Rundreisen durch diese Städte als Atome auf der Erde. Würden wir alle möglichen Rundreisen aufzählen und jedesmal ihre Länge berechnen, dann fielen

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

viele Additionen an (die Approximation der Fakultät ist nicht trivial, sie ist unter dem Namen *Stirling-Formel* bekannt). Bereits vor langer Zeit wurde ein besserer Algorithmus entdeckt, der ungefähr 2^n viele Operationen ausführt. Mit diesem konnten im Jahr 1970 Rundreisen gefunden werden, die bis zu $n = 120$ Städte umfassten. Würde man diesen Algorithmus heute auf einem aktuellen Rechner ausführen, dann könnten nur Rundreisen durch maximal $n = 140$ Städte gefunden werden. Tatsächlich aber können heute Rundreisen berechnet werden, die mehr als $100'000$ Städte besuchen. Diese enorme Verbesserung wurde hauptsächlich durch Fortschritte in der Algorithmik ermöglicht, nicht aber durch alleinige Erhöhung der Rechenleistung. Dieses Beispiel zeigt eindrucksvoll, wie wichtig die Entwicklung effizienter Algorithmen ist.

1.2 Beispiele des Algorithmenentwurfs

1.2.1 Multiplikation ganzer Zahlen

Bereits in der Primarschule haben wir einen einfachen Algorithmus kennengelernt, nämlich denjenigen zur Multiplikation zweier Zahlen. Gegeben seien dabei zwei Zahlen z_1 und z_2 in Dezimaldarstellung, und gesucht ist das Produkt $z_1 \cdot z_2$ dieser beiden Zahlen (ebenfalls in Dezimaldarstellung). Dabei multiplizieren wir jede Ziffer der einen Zahl mit jeder Ziffer der anderen Zahl, und summieren die (entsprechend nach links verschobenen) Teilprodukte auf, um das Endergebnis zu erhalten. Abbildung 1.2 zeigt beispielhaft die Multiplikation der Zahlen 62 und 37 nach dieser Methode.

MULTIPLIKATION: PRIMARSCHULMETHODE

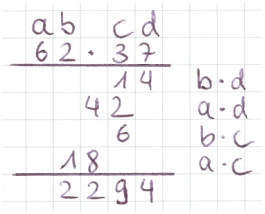


Abb. 1.2 Multiplikation von 62 und 37 nach der Primarschulmethode.

Algorithmus von Karatsuba und Ofman Die Primarschulmethode benötigt zur Multiplikation zweier zweistelliger Zahlen also $2 \cdot 2 = 4$ Multiplikationen von Ziffern. Es ist leicht zu sehen, dass die Methode zur Multiplikation zweier n-stelliger Zahlen n^2 einstellige Multiplikationen benötigt. Karatsuba und Ofman schlugen 1962 ein verbessertes Verfahren vor, das mit weniger einstelligen Multiplikationen auskommt. Die Kosten für die Additionen ignorieren wir für den Moment.

ANZAHL EINSTELLIGER MULTIPLIKATIONEN

Um nun mit weniger Multiplikationen auszukommen, beobachten wir, dass für zwei zweistellige Zahlen $z_1 = 10a + b$ und $z_2 = 10c + d$

VERBESSERUNG

$$(10a + b) \cdot (10c + d) = 100a \cdot c + 10a \cdot c + 10b \cdot d + b \cdot d + 10(a - b) \cdot (d - c)$$

gilt (nachrechnen!). Schaut man diesen Ausdruck genau an, wird man feststellen, dass hier nur noch drei verschiedene Produkte zweier Ziffern vorhanden sind, nämlich $a \cdot c$, $b \cdot d$, und $(a - b) \cdot (d - c)$. Ein kritischer Leser wird nun möglicherweise einwenden, dass auch die Multiplikation mit 10 oder 100 eine Multiplikation ist. Das stimmt natürlich, wir zählen Multiplikationen mit Zehnerpotenzen aber dennoch nicht, da sie verhältnismässig leicht realisiert werden können: Wird eine Zahl z mit 10^k multipliziert, müssen an die Dezimaldarstellung von z lediglich k Nullen angehängt werden. Wir haben also eine Methode gefunden, zwei zweistellige Zahlen mit lediglich drei einstelligen Multiplikationen auszurechnen. Abbildung 1.3 zeigt erneut die Multiplikation der Zahlen 62 und 37, diesmal nach der verbesserten Methode.

In der Informatik, z.B. in der Kryptographie, braucht man heute häufig sehr grosse Zahlen. Es stellt sich die Frage, ob die obige Methode auch zur Multiplikation zweier Zahlen mit mehr als zwei Ziffern benutzt werden kann. Betrachten wir zum Beispiel das Produkt $6237 \cdot 5898$. Wir beobachten nun, dass wir die Faktoren als zwei zweistellige Zahlen mit den "Ziffern" 62, 37, 58, 98 auffassen und dann *rekursiv* bzw. *induktiv* dieselbe Methode anwenden können. In diesem Beispiel würde unsere neue Methode 9 einstellige Multiplikationen benötigen, während die Schulmethode 16(=

GRÖSSERE ZAHLEN

INDUKTIVES PRINZIP

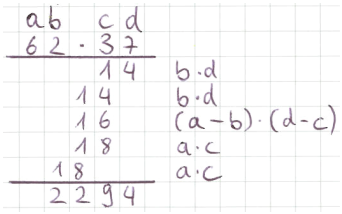


Abb. 1.3 Multiplikation von 62 und 37 nach der Methode von Karatsuba und Ofman. Das oberste Teilprodukt entspricht $b \cdot d$ (und ist daher nicht nach links verschoben), die mittleren Teilprodukte entsprechen $10(b \cdot d)$, $10(a-b) \cdot (d-c)$ und $10a \cdot c$ (und sind daher um eine Stelle nach links verschoben), das unterste entspricht $100a \cdot c$ (und ist daher um zwei Stellen nach links verschoben).

$4 \cdot 4$) benötigt. Dieses Prinzip der induktiven Anwendung kommt in der Algorithmik sehr häufig vor.

VERALLGEMEINERUNG

Haben wir allgemein zwei n-stellige Zahlen (wobei wir vereinfachend annehmen wollen, dass $n = 2^k$ eine Zweierpotenz ist), dann können wir diese als $10^{n/2}a + b$ bzw. $10^{n/2}c + d$ schreiben. Wie früher beobachten wir nun, dass

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d)& \\ &= 10^n a \cdot c + 10^{n/2}a \cdot c + 10^{n/2}b \cdot d + b \cdot d + 10^{n/2}(a-b) \cdot (d-c)\end{aligned}$$

ANALYSE

gilt. Die Produkte $a \cdot c$, $b \cdot d$ und $(a-b) \cdot (d-c)$ berechnen wir dann rekursiv. Wir analysieren nun das Verfahren genauer, um zu untersuchen, wie viele einstellige Multiplikationen es ausführt. Dazu definieren wir $M(n)$ als die Anzahl einstelliger Multiplikationen bei zwei Zahlen mit je $n = 2^k$ Ziffern mit unserer neuen Methode. Wir haben gesehen, dass wir zwei einstellige Zahlen mit einer elementaren Multiplikation multiplizieren können, und zwei zweistellige Zahlen mit drei elementaren Multiplikationen. Allgemein erhalten wir

$$M(2^k) = \begin{cases} 1 & \text{falls } k = 0 \text{ ist} \\ 3 \cdot M(2^{k-1}) & \text{falls } k > 0 \text{ ist.} \end{cases} \tag{1}$$

TELESKOPIEREN

Um nun die Rekursionsgleichung (1) aufzulösen, *teleskopieren* wir, d.h., wir setzen die Rekursionsformel einige Male ein, bis wir eine Vermutung für die explizite Formel erhalten:

$$M(2^k) = 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) = \cdots \stackrel{!}{=} 3^k \cdot M(2^0) = 3^k.$$

Wir vermuten also, dass $M(2^k) = 3^k$ gilt. Dies beweisen wir nun mittels *vollständiger Induktion* (die in Abschnitt 1.4.1 noch genauer besprochen wird):

INDUKTIONSBEWEIS

Induktionsvermutung: Wir vermuten, dass $M(2^k) = 3^k$ gilt.

Induktionsanfang ($k = 0$): Es ist $M(2^0) = 3^0 = 1$, also ist die Induktionsvermutung für $k = 0$ korrekt.

Induktionsschritt ($k \to k+1$): Für $k > 1$ gilt $M(2^{k+1}) \stackrel{\text{Def.}}{=} 3M(2^k) \stackrel{I.V.}{=} 3 \cdot 3^k = 3^{k+1}$ (*Hinweis: I.V.* steht hier abkürzend für *Induktionsvermutung*). Damit ist die Aussage für alle k korrekt. ∎

Wir hatten früher argumentiert, dass die Primarschulmethode zur Multiplikation zweier n-stelliger Zahlen n^2 einstellige Multiplikationen ausführt. Um zu sehen, wie viele solche Operationen die Methode von Karatsuba und Ofman benötigt, ersetzen wir in der eben berechneten Formel 2^k durch n (und k durch $\log_2(n)$) und erhalten

VERBESSERUNG

$$M(n) = 3^{\log_2 n} = \left(2^{\log_2 3}\right)^{\log_2 n} = 2^{(\log_2 3)(\log_2 n)} = n^{\log_2 3} \approx n^{1.58}, \qquad (2)$$

was bedeutend besser als die Primarschulmethode ist. Für grosse Zahlen ist der Algorithmus von Karatsuba und Ofman also schnell um ein Vielfaches schneller als die Primarschulmethode. Als konkretes Beispiel: Für zwei tausendstellige Zahlen ist unser neues Verfahren $\frac{1000^2}{1000^{1.58}} \approx 18$ Mal schneller als die Primarschulmethode.

UNTERE SCHRANKE

Im Idealfall möchte man zusätzlich häufig gerne noch eine theoretische untere Schranke finden, um zu zeigen, dass es gar nicht schneller gehen kann. Man kann sich die Frage stellen, wie viele einstellige Multiplikationen mindestens notwendig sind, um zwei n-stellige Zahlen zu multiplizieren. Diese Frage ist noch nicht endgültig geklärt. Man weiss jedoch, dass es mindestens n elementare Multiplikationen sein müssen. Ein nicht ganz präzises Argument dafür: Würden wir weniger als $\frac{n}{2}$ einstellige Multiplikationen durchführen, so könnten wir erst gar nicht alle Ziffern der Eingabe anschauen.

1.2.2 Star finden

Wie im vorigem Abschnitt gesehen gehen Algorithmenentwurf und dessen Analyse Hand in Hand. Wäre eine Übungsaufgabe gewesen "Finde ein schnelleres Verfahren für die Multiplikation zweier Zahlen", wäre es schwierig gewesen, die Methode von Karatsuba einfach so zu finden. Häufiger ist der Algorithmenentwurf aber eine sehr systematische, nachvollziehbare Sache. Dazu betrachten wir ein weiteres Beispiel.

PROBLEMBESCHREIBUNG

Gegeben sei ein Raum mit n Personen. Gesucht ist ein *Star*. Ein Star ist eine Person, den alle im Raum kennen, und der selber niemanden anders kennt. Wir erlauben nur eine einzige elementare Operation, nämlich eine Frage an eine beliebige Person A, ob sie eine andere Person B kennt. Als mögliche Antworten sind nur "Ja" und "Nein" erlaubt. Andere Fragen sind nicht erlaubt. Wir möchten nun mit möglichst wenigen Fragen ermitteln, ob sich im Raum ein Star befindet.

PROBLEMEIGENSCHAFTEN

Bevor wir uns überlegen, wie das gehen könnte, überlegen wir zunächst, welche Eigenschaften das Problem hat. Wir beobachten:

- Es kann sein, dass es keinen Star gibt (z.B. wenn jeder jeden anderen kennt).
- Es kann sein, dass es genau einen Star gibt (z.B. wenn George Clooney in den Raum käme).
- Es kann nicht mehr als einen Star geben. Angenommen, es gäbe zwei Stars S_1 und S_2. Nun es gibt es zwei Möglichkeiten: Entweder, S_1 kennt S_2, oder nicht. Im ersten Fall wäre S_1 kein Star, ansonsten wäre S_2 kein Star.

NAIVE LÖSUNG

Algorithmus 1 (Naiv) Eine naive Strategie zur Lösung des Problems besteht darin, jeden über jeden anderen auszufragen. Wir erzeugen eine Tabelle mit n Zeilen und n Spalten, und tragen in dem Eintrag in Zeile A und Spalte B genau dann "Ja" ein, wenn die Person A die Person B kennt, und "Nein" sonst. Die Diagonalelemente können wir ignorieren, da wir annehmen, dass jeder Mensch sich selbst kennt. Wie finden wir nun den Star in dieser Tabelle? Wir suchen eine Person, sodass ihre Spalte

	1	2	3
1	-	Ja	Nein
2	Nein	-	Nein
3	Ja	Ja	-

Abb. 1.4 Beispiel für eine Situation, in der ein Star existiert, nämlich 2.

nur "Ja" enthält (alle kennen sie) und ihre Zeile nur aus "Nein" besteht (sie kennt niemanden). Abbildung 1.4 zeigt ein Beispiel für eine solche Situation: Person 2 ist ein Star.

Anzahl gestellter Fragen

Ein Nachteil dieses naiven Verfahrens ist, dass sehr viele Fragen gestellt werden, nämlich $n \cdot (n-1)$ (also alle möglichen). Bei der Multiplikation zuvor haben wir argumentiert, dass es nicht besser gehen kann, als jede Ziffer mindestens einmal anzuschauen. Hier ist das ein bisschen anders: Es ist nicht ausgeschlossen, dass wir den Star finden können oder mit Sicherheit sagen können, dass es keinen Star gibt, ohne jede mögliche Frage zu stellen.

Induktive Lösung

Algorithmus 2a (Induktiv) Im vorigen Abschnitt hat es uns geholfen, das Problem in kleinere Teile zu zerlegen, d.h., es induktiv zu lösen. Wenn es im Raum $n = 2$ Personen gibt, dann können wir immer einen Star mit $F(2) = 2$ Fragen finden. Gibt es im Raum $n > 2$ Person, dann könnten wir wie folgt vorgehen: Wir schicken eine Person nach draussen, bestimmen rekursiv den potentiellen Star unter den verbleibenden Personen und holen die abwesende Person wieder in den Raum. Für diese Person müssen wir prüfen, ob sie der Star ist, was $2(n-1)$ Fragen kosten kann. Damit werden im schlimmsten Fall insgesamt $F(n) = 2(n-1) + F(n-1) = 2(n-1) + 2(n-2) + \cdots + 2 = n(n-1)$ viele Fragen gestellt, was leider noch keine Verbesserung gegenüber dem naiven Verfahren darstellt.

Verbesserte Lösung

Algorithmus 2b (Verbesserung) Wieso sparen wir keine Fragen? Das Problem besteht darin, dass die herausgeschickte Person genau der Star sein kann. Dann nämlich brauchen wir viele Fragen, wenn er den Raum wieder betritt. Wir müssten also irgendwie garantieren, dass wir nicht den Star aus dem Raum schicken. Dies ist aber einfach machbar: Wir fragen eine beliebige Person A im Raum, ob sie eine beliebige andere Person B im Raum kennt. Falls ja, dann ist A kein Star, ansonsten ist B kein Star. Wenn die zuvor herausgeschickte Person nun den Raum wieder betritt, dann reichen zwei weitere Fragen um herauszufinden, ob der ggf. im Raum gefundene potentielle Star wirklich ein Star ist. Für die Anzahl der maximal gestellten Fragen ergibt sich also

Analyse

$$F(n) = \begin{cases} 2 & \text{für } n = 2 \\ 1 + F(n-1) + 2 & \text{für } n > 2. \end{cases} \tag{3}$$

Wie zuvor teleskopieren wir und erhalten

$$F(n) = 3 + F(n-1) = 3 + 3 + F(n-2) = \cdots = 3(n-2) + 2 = 3n - 4, \tag{4}$$

was nun noch mit vollständiger Induktion über n bewiesen werden muss (Übung!). Wir sehen, dass in unserem neuen Verfahren deutlich weniger Fragen als im naiven Verfahren gestellt werden.

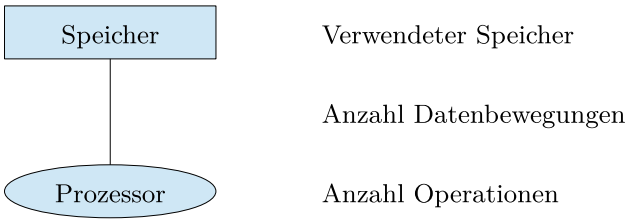


Abb. 1.5 Ein stark vereinfachtes Rechnermodell mit drei Komponenten: Prozessor, Speicher und ein Bus, der Prozessor und Speicher miteinander verbindet.

1.3 Kostenmodell

Um die Effizienz von Algorithmen formal zu untersuchen, müssen wir zunächst überlegen, wie wir ihre Kosten bestimmen. Dazu definieren ein stark vereinfachtes Rechnermodell, das lediglich aus einem *Prozessor*, einem (unbeschränkt grossen) *Speicher* sowie einem *Bus*, der Prozessor und Speicher miteinander verbindet, besteht. Mögliche Kostenmasse wären nun z.B. die Anzahl vom Prozessor ausgeführter Operationen, der benutzte Speicher, oder die Anzahl der Datenbewegungen zwischen dem Prozessor und dem Speicher. Beispiele für die vom Prozessor ausführbaren *elementaren Operationen* sind

RECHNERMODELL

ELEMENTAROPERATION

- *Rechenoperationen* wie die Addition, Subtraktion, Multiplikation oder Division zweier natürlicher Zahlen,
- *Vergleichsoperationen* wie z.B. “<”, “>” oder “=” zwischen zwei natürlichen Zahlen,
- *Zuweisungen* der Form $x \leftarrow A$, wo der Variable x auf der linken Seite der Zahlenwert des Ausdrucks A auf der rechten Seite zugewiesen wird.

Weiter ins Detail, auf die Ebene der einzelnen Bits und Bytes, wollen wir in dieser Vorlesung meist nicht gehen. Die entscheidende Idee ist also, zu abstrahieren. Wie lange eine einzelne der obengenannten Operationen effektiv dauert, hängt von vielen Faktoren ab: der Taktzahl unseres Prozessors, der Effizienz des Compilers, der Ausnutzung des Cache-Speichers, der Wahl der Programmiersprache usw. In jedem realen Rechner ist die Laufzeit dieser Operationen aber schlimmstenfalls konstant. Eine erste Idee besteht nun darin, konstante Faktoren ignorieren, da diese von Maschine zu Maschine variieren. Wir weisen nun vereinfachend jeder der obengenannten Operationen die Kosten 1 zu, und sprechen daher auch vom *Einheitskostenmodell*. Dieses nimmt also an, dass der Aufwand für arithmetische Operationen unabhängig von der Grösse der Operanden ist.

EINHEITSKOSTENMODELL

1.3.1 $\mathcal{O}$-Notation

Eine weitere Idee ist, dass wir uns nur für grosse Eingabewerte interessieren, bei denen der Algorithmus lange läuft, denn bei kleinen Eingaben ist auch ein naiver Algorithmus vernünftig schnell. Wollen wir zwei Algorithmen miteinander vergleichen, dann ist derjenige besser, der auf grossen Eingaben schneller läuft. Darüber hinaus interessiert uns nur das Wachstum einer Funktion. Um dies mathematisch präzise zu beschreiben, wollen wir erneut vereinfachend alle konstanten Faktoren ignorieren.

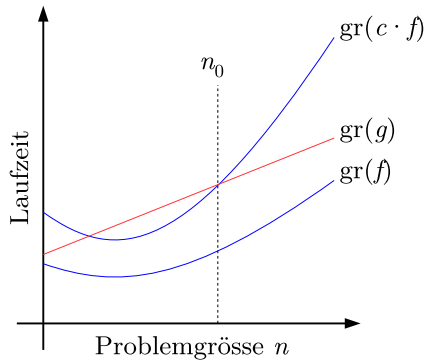


Abb. 1.6 Die Funktion g liegt in $\mathcal{O}(f)$, denn für alle $n \geq n_0$ ist $g(n) \leq cf(n)$.

Eine lineare Wachstumskurve mit Steigung 1 erachten wir als gleich schnell wie eine Kurve mit Steigung 20.

$\mathcal{O}$-Notation

Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Kostenfunktion. Wir definieren nun eine Menge von Funktionen, die auch nicht schneller wachsen als f, konkret

$$\mathcal{O}(f) := \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \begin{array}{l} \text{es gibt } c > 0, n_0 \in \mathbb{N} \text{ so, dass } g(n) \leq cf(n) \\ \text{für alle } n \geq n_0 \end{array} \right\}. \quad (5)$$

Anschaulich bedeutet dies, dass f (bis auf einen konstanten Faktor) asymptotisch gesehen eine obere Schranke für g bildet (siehe Abbildung 1.6). Diese $\mathcal{O}$-Notation (auch *$\mathcal{O}$-Kalkül* oder *Bachmann-Landau-Notation* genannt) erlaubt uns nun, Ausdrücke zu vereinfachen und nach oben abzuschätzen. So ist z.B. $3n - 4 \in \mathcal{O}(n)$. Dies

Beispiele

kann man sehen, indem man etwa $c = 3$ und $n_0 = 1$ wählt. Weitere Beispiele sind

- $2n \in \mathcal{O}(n^2)$ (mit $c = 2$ und $n_0 = 1$),
- $n^2 + 100n \in \mathcal{O}(n^2)$ (mit $c = 2$ und $n_0 = 100$),
- $n + \sqrt{n} \in \mathcal{O}(n)$ (mit $c = 2$ und $n_0 = 1$).

Nutzen

Die $\mathcal{O}$-Notation erlaubt es uns also nicht nur, alle Konstanten wegzulassen, wir können auch Terme niedrigerer Ordnung ignorieren. So ist z.B. auch $n^2 + n^{1,9} + 17\sqrt{n} + 3\log(n) \in \mathcal{O}(n^2)$. Wir können also für jeden Term den einfachsten Vertreter wählen. Dies ist sinnvoll, da uns, wie vorher erwähnt, nicht die genaue Kostenfunktion, sondern lediglich ihr Wachstumsverhalten (z.B. linear, quadratisch, kubisch, usw.) für grosse n interessiert.

Addition

Die $\mathcal{O}$-Notation hat einige angenehme Eigenschaften, z.B. abgeschlossen unter Additionen zu sein: Sind sowohl $g_1 \in \mathcal{O}(f)$ als auch $g_2 \in \mathcal{O}(f)$, dann ist auch $g_1 + g_2 \in \mathcal{O}(f)$. Dies sieht man wie folgt: Ist $g_1 \in \mathcal{O}(f)$, dann gibt es Konstanten $c_1 > 0$ und n_1, sodass $g_1(n) \leq c_1 f(n)$ für alle $n \geq n_1$ gilt. Analog gibt es Konstanten $c_2 > 0$ und n_2, sodass $g_2(n) \leq c_2 f(n)$ für alle $n \geq n_2$ gilt. Wir wählen nun $c = c_1 + c_2$ sowie $n_0 = \max\{n_1, n_2\}$, und sehen dass dann $g_1(n) + g_2(n) \leq c_1 f(n) + c_2 f(n) = cf(n)$ für alle $n \geq n_0$ gilt, was unsere Aussage beweist.

Ω-Notation

Die $\mathcal{O}$-Notation benutzen wir, um obere Schranken anzugeben. Für untere Schranken definieren wir analog

$$\Omega(f) := \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \begin{array}{l} \text{es gibt } c > 0, n_0 \in \mathbb{N} \text{ so, dass } g(n) \geq cf(n) \\ \text{für alle } n \geq n_0 \end{array} \right\}. \quad (6)$$

Anschaulich bedeutet dies, dass g asymptotisch mindestens so schnell wie f wächst. Liegt eine Funktion g sowohl in $\mathcal{O}(f)$ als auch in $\Omega(f)$, dann sagen wir, dass sie asymptotisch genauso schnell wie f wächst. Wir schreiben dann auch $f \in \Theta(g)$, und definieren

Θ-Notation

$$\Theta(f) := \mathcal{O}(f) \cap \Omega(f). \tag{7}$$

Um die Konzepte zu vertiefen, betrachten wir nun einige weitere Beispiele.

Beispiele

- $n \in \mathcal{O}(n^2)$: Diese Aussage ist korrekt, aber ungenau. Tatsächlich gelten ja auch $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$: Auch diese Aussage ist korrekt, man würde aber üblicherweise $\mathcal{O}(n^2)$ statt $\mathcal{O}(2n^2)$ schreiben, da Konstanten in der $\mathcal{O}$-Notation weggelassen werden können.
- $2n^2 \in \mathcal{O}(n)$: Diese Aussage ist falsch, da $\frac{2n^2}{n} = 2n \xrightarrow[n\to\infty]{} \infty$ und daher durch keine *Konstante* nach oben beschränkt werden kann.
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$: Diese Aussage ist korrekt, da alle Funktionen, die höchstens linear wachsen, auch nur höchstens quadratisch wachsen. Mathematisch falsch wäre die Aussage $\mathcal{O}(n) \in \mathcal{O}(n^2)$, da $\mathcal{O}(n)$ eine *Menge* und kein Element ist.
- $\Theta(n) \subseteq \Theta(n^2)$: Diese Aussage ist falsch, denn alle Funktionen in $\Theta(n^2)$ erfüllen auch $\Omega(n^2)$. Andererseits ist enthält $\Theta(n)$ Funktionen wie $f(n) = n$, die nicht in $\Omega(n^2)$ liegen.

Nützlich zur Untersuchung, in welcher Beziehung zwei Funktionen f und g zueinander stehen, ist das folgende Theorem.

$\mathcal{O}$-Notation und Grenzwerte

Theorem 1.1. *Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ zwei Funktionen. Dann gilt:*

1) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dann ist $f \in \mathcal{O}(g)$, und $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.

2) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (wobei C konstant ist), dann ist $f \in \Theta(g)$.

3) Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, dann ist $g \in \mathcal{O}(f)$, und $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

Beweis. Wir beweisen nur die erste Aussage. Der Beweis der übrigen Aussagen verläuft analog. Ist $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dann gibt es für *jedes* $\varepsilon > 0$ ein n_0, sodass $\frac{f(n)}{g(n)} < \varepsilon$ für alle $n \geq n_0$ gilt. Wir können also ein *beliebiges* $\varepsilon > 0$ wählen, und die Definition des Grenzwerts garantiert uns, dass ein geeignetes n_0 existiert, sodass $f(n) \leq \varepsilon g(n)$ für alle $n \geq n_0$ gilt. Damit folgt direkt $f \in \mathcal{O}(g)$.

Wir wissen daher auch sofort, dass $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ gilt. Angenommen, es wäre zusätzlich $\mathcal{O}(f) = \mathcal{O}(g)$. Dann wäre natürlich auch $g \in \mathcal{O}(f)$. Also gäbe es ein $c > 0$ und ein n_0, sodass $g(n) \leq cf(n)$ für alle $n \geq n_0$ gilt. Dann wäre aber

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq \lim_{n\to\infty} \frac{1}{c} = \frac{1}{c} > 0,$$

was ein Widerspruch zur Voraussetzung ist. Also ist $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$. ∎

Schreibweise

Üblicherweise schreibt man vereinfachend $f = O(g)$ anstatt $f \in \mathcal{O}(g)$. Mathematisch gesehen ist dies natürlich falsch, da f eine Funktion und $\mathcal{O}(g)$ eine Menge bezeichnet. Ausserdem folgt aus $f_1 = O(g)$ und $f_2 = O(g)$ nicht notwendigerweise $f_1 = f_2$. So zum Beispiel sind sowohl $n = O(n^2)$ als auch $n^2 = O(n^2)$, aber natürlich ist $n \neq n^2$. Die Notation ist eher im Sinne des umgangssprachlichen "ist ein" zu verstehen: Eine Kuh ist ein Tier, aber nicht jedes Tier ist ein Kuh. Gleichungen der Form $f = O(g)$ müssen also von links nach rechts gelesen werden, und weiter rechts wird es möglicherweise ungenauer. So zum Beispiel ist $2n^2 + 1 = O(2n^2) = O(n^2)$, und wir erlauben sogar Schreibweisen wie $5n + 7 = O(n) = O(n^2)$. Der zweite Gleichheitszeichen muss also als Teilmengenzeichen interpretiert werden. Diese vereinfachte Schreibweise ist Tradition und hat sich als praktisch erwiesen. Wir erlauben diesen Missbrauch der Notation, wenn aus dem Kontext heraus völlig klar ist, was gemeint ist. Dies heisst aber auch, dass im Zweifelsfall auf jeden Fall die präzise Mengen-basierte Notation verwendet werden sollte!

1.3.2 Komplexität, Kosten und Laufzeit

Probleme, Algorithmen und Programme

Wir haben bereits früher gesehen, dass *Algorithmen* verwendet werden, um *Probleme* zu lösen. Es gibt nun viele Arten, Algorithmen zu beschreiben, etwa textlich (wie wir das beim einfachen Algorithmus zur Lösung des Kuhproblems getan haben), bildlich (z.B. durch Flussdiagramme), durch Pseudocode, oder eben durch richtige *Programme*. Als *Programm* bezeichnen wir Code, der auf einem konkreten Rechner (ggf. kompiliert und dann) ausgeführt werden kann. Ein Programm ist also eine konkrete Implementierung eines Algorithmus in einer Programmiersprache.

Kosten

Laufzeit

Beziehung Kosten und Laufzeit

Für jeden Algorithmus können wir nun (zumindest theoretisch) seine *Kosten*, d.h., die Anzahl der durchgeführten elementaren Operationen, bestimmen. Analog können wir für ein Programm seine *Laufzeit* (z.B. in Sekunden) messen, wenn es auf einem realen Rechner ausgeführt wird. Gibt es nun eine eindeutige Beziehung zwischen den Kosten eines Algorithmus A und der Laufzeit eines Programms P, das A implementiert? Offensichtlich ist die Laufzeit von P nicht einfach nur ein Vielfaches der Kosten von A, da die Laufzeiten von einer Operation nicht immer gleich sind. Denken wir etwa an die Addition zweier Zahlen: Befinden sich diese bereits im Speicher, dann kann die Operation vermutlich schneller ausgeführt werden, als wenn die Zahlen erst von einem Externspeicher (z.B. einer langsamen Festplatte) gelesen werden müssten. Wir beobachten aber auch, dass auf einem konkreten Rechner die Laufzeit jeder Operation durch eine Konstante sowohl nach unten, als auch nach oben beschränkt werden kann. Prozessorkerne in modernen Rechnern arbeiten oft mit 3GHz oder mehr, d.h., es gibt $3 \cdot 10^9$ Takte pro Sekunde. Pro Takt können eine bestimmte Anzahl Operationen ausgeführt werden (z.B. 8), also gibt es eine untere Schranke. Auch eine obere Schranke können wir angeben: Mit Sicherheit dauert eine einzelne Operation auf einem realen Rechner niemals länger als eine Stunde. Diese Konstanten für die untere und die obere Schranke liegen natürlich sehr weit auseinander. Wir beobachten aber, dass die Kosten eines Algorithmus und die Laufzeit einer Implementierung *asymptotisch gesehen* übereinstimmen.

Können wir von den Kosten eines Algorithmus auch auf die Schwierigkeit des zugrundeliegenden Problems schliessen? Ja, aber wir erhalten lediglich eine obere Schranke. Es könnte nämlich andere Algorithmen geben, die wir noch nicht kennen, und die das Problem mit geringeren Kosten lösen. Wir definieren also die *Komplexität* eines Problems P als die minimalen Kosten über alle Algorithmen A, die P

Komplexität

Problem	Komplexität	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
		$\uparrow$	$\uparrow$	$\uparrow$
Algorithmus	Kosten (# el. Op.)	$3n-4$	$\mathcal{O}(n)$	$\Theta(n^2)$
		$\downarrow$	$\updownarrow$	$\updownarrow$
Programm	Laufzeit (Sek.)	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$

Abb. 1.7 Beziehungen zwischen der Komplexität eines Problems, den Kosten eines Algorithmus für dieses Problem und der Laufzeit einer Implementierung von diesem Algorithmus.

lösen. Da die Menge aller Algorithmus unendlich gross ist, können wir nur für wenige Probleme die Komplexität exakt angeben. Die Komplexität der Multiplikation zweier Zahlen der Länge n ist offenbar in $\Omega(n)$, denn jede Ziffer der Eingabe muss berücksichtigt werden. Ebenso ist sie in $\mathcal{O}(n^{\log_3(2)})$, denn der Algorithmus von Karatsuba und Ofman hat genau diese Kosten. Abbildung 1.7 zeigt die Beziehungen zwischen den definierten Begriffen.

1.4 Mathematische Grundlagen

1.4.1 Beweise per Induktion

In diesem Abschnitt wird eine sehr wichtige Beweistechnik eingeführt, nämlich den Beweis per *Induktion*. Beim klassischen Induktionsbeweis geht es darum, zu zeigen, dass eine Aussage für alle natürlichen Zahlen $n \in \mathbb{N}$ Gültigkeit besitzt. Eine Induktion besteht aus den folgenden Komponenten:

INDUKTIONSBEWEISE

1) *Induktionsanfang*: Zeige, dass die Aussage für $n = 1$ gilt.

2) *Induktionshypothese*: Wir nehmen an, die Aussage sei gültig für ein allgemeines $n \in \mathbb{N}$.

3) *Induktionsschritt*: Zeige, dass aus der Gültigkeit der Aussage für n (Induktionshypothese) die Gültigkeit der Aussage für $n+1$ folgt.

Eine verallgemeinerte Variante der Induktion erlaubt eine stärkere Induktionshypothese, nämlich dass die Aussage gültig *für alle* $k \leq n$ sei (aus der dann natürlich ebenfalls im Induktionsschritt die Gültigkeit der Aussage für $n+1$ geschlussfolgert werden muss). Ebenso kann der Induktionsanfang für eine Zahl $n_0 > 1$ erfolgen (dann gilt die Aussage aber nicht mehr für alle $n \in \mathbb{N}$, sondern lediglich für alle $n \in \mathbb{N}$ mit $n \geq n_0$).

VERALLGEMEINERUNG

Beispiel 1 Wir möchten zeigen, dass $(1+x)^n \geq 1+nx$ für alle $n \in \mathbb{N}$ und alle $x \geq -1$ gilt (dies ist die sog. *Bernoulli-Ungleichung*).

BERNOULLI-UNGLEICHUNG

Induktionsanfang ($n = 1$): Es ist $(1+x)^1 = 1+x \geq 1+1 \cdot x$, also ist die Behauptung für $n = 1$ wahr.

Induktionshypothese: Angenommen, es gelte $(1+x)^n \geq 1+nx$ für ein $n \in \mathbb{N}$.

Induktionsschritt ($n \to n+1$): Wir beobachten, dass $(1+x)^{n+1} = (1+x)^n(1+x)$ gilt, und schätzen $(1+x)^n$ mittels der Induktionshypothese nach oben ab.

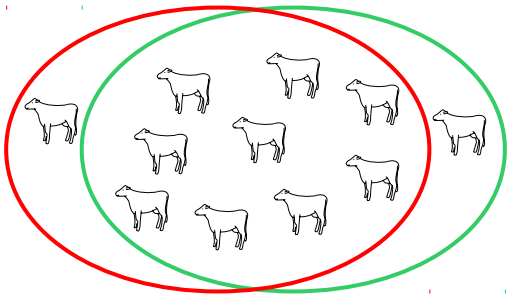


Abb. 1.8 Eine Kuhherde mit $n = 11$ Kühen.

Konkret erhalten wir also

$$\begin{aligned}(1+x)^{n+1} &= (1+x)^n(1+x)\\ &\overset{I.H.}{\geq} (1+nx)(1+x)\\ &= 1+x+nx+nx^2\\ &\geq 1+x+nx = 1+(n+1)x. \quad \blacksquare\end{aligned}$$

Beispiel 2 Induktionsbeweise müssen gründlich geführt werden, da man sonst leicht auch falsche Aussagen "beweisen" kann, wie das folgende Beispiel zeigt. Jemand behauptet "Alle Kühe einer Kuhherde haben die gleiche Farbe" und zeigt den folgenden "Induktionsbeweis", um seine Behauptung zu untermauern.

Induktionsanfang ($n = 1$): Wir betrachten eine Kuhherde mit genau einer Kuh. Diese hat trivialerweise die gleiche Farbe wie alle anderen Kühe in der Herde (es gibt keine), folglich ist die Behauptung für $n = 1$ wahr.

Induktionshypothese: Angenommen, alle Kühe in einer Herde mit n Kühen hätten die gleiche Farbe.

Induktionsschritt ($n \to n+1$): Betrachte eine Herde mit $n+1$ Kühen. Wir können nun eine Kuh aus der Herde entfernen und erhalten eine Herde mit n Kühen. In Abbildung 1.8 sind das alle Kühe im roten Kreis. Diese haben gemäss Induktionshypothese alle die gleiche Farbe. Nun führen wir die entfernte Kuh wieder der Herde zu und entfernen dafür eine andere Kuh aus der Herde. Wieder erhalten wir eine Herde mit n Kühen. In Abbildung 1.8 sind das alle Kühe im grünen Kreis. Da auch diese gemäss Induktionshypothese alle die gleiche Farbe haben und sich die Herden im roten und im grünen Kreis überschneiden, haben alle $n+1$ Kühe der ursprünglichen Herde die gleiche Farbe. ▮

Der obige Beweis ist offensichtlich inkorrekt, aber warum? Das Problem liegt hier im Induktionsschritt. Wir nehmen an, dass der Schnitt des grünen und des roten Kreises immer mindestens eine Kuh enthält. Dies ist aber für $n = 2$ nicht der Fall. Tatsächlich liegt genau hier das Problem: Für $n = 2$ Kühe unterteilen wir die Herde in zwei Herden mit je einer Kuh, die nicht notwendigerweise die gleiche Farbe haben. Könnten wir beweisen, dass je zwei Kühe *immer* die gleiche Farbe haben, dann wäre der obige Beweis korrekt.

1.4.2 Summenformeln

In diesem Abschnitt werden wir einige Summen der Form $\sum_{k=1}^{n} a_k$ kennenlernen und untersuchen, wie sie aufgelöst werden können. Sind zum Beispiel $a_1 = \cdots = n_k = 1$, dann ist relativ leicht ersichtlich, dass

$$\sum_{k=1}^{n} n^0 = \sum_{k=1}^{n} 1 = 1 + \cdots + 1 = n \tag{8}$$

$\sum_{k=1}^{n} 1$

gilt. Weniger offensichtlich, aber vielen sicherlich bekannt, ist die Formel

$$S_n = \sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + n. \tag{9}$$

$\sum_{k=1}^{n} k$

Die Korrektheit einer solchen Formel kann man (fast immer) mit vollständiger Induktion über n beweisen. Bei der obigen Formel (9) können wir noch einen weiteren Trick anwenden: Wir schreiben die Summanden einmal vorwärts und direkt darunter in umgekehrter Reihenfolge auf. Wir beobachten

$$\begin{array}{ccccccccc} 1 & + & 2 & + & \cdots & + & n & = & S_n \\ n & + & n-1 & + & \cdots & + & 1 & = & S_n \\ \hline n+1 & & n+1 & & \cdots & & n+1 & = & n(n+1), \end{array}$$

d.h. $S_n + S_n = n(n+1)$, also ist $S_n = \frac{n(n+1)}{2}$.

Wie finden wir aber eine geschlossene Form für andere Summen? Dazu betrachten wir als weiteres Beispiel die Summe

$$S'_n = \sum_{k=1}^{n} k^2 = 1^2 + 2^2 + \cdots + n^2. \tag{10}$$

$\sum_{k=1}^{n} k^2$

Idee 1

1) Vermute, S'_n sei von der Form $an^3 + bn^2 + cn + d$.

2) Setze $n = 1, 2, 3, 4$ ein und erhalte ein Gleichungssystem mit vier Gleichungen und vier Unbekannten:

$$\begin{aligned} a \cdot 1^3 + b \cdot 1^2 + c \cdot 1^1 + d \cdot 1^0 &= 1 = 1^2 = S_1 \\ a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \cdot 2^0 &= 5 = 1^2 + 2^2 = S_2 \\ a \cdot 3^3 + b \cdot 3^2 + c \cdot 3^1 + d \cdot 3^0 &= 14 = 1^2 + 2^2 + 3^2 = S_3 \\ a \cdot 4^3 + b \cdot 4^2 + c \cdot 4^1 + d \cdot 4^0 &= 30 = 1^2 + 2^2 + 3^2 + 4^2 = S_4. \end{aligned}$$

Mit Methoden, die wir bald in linearer Algebra lernen, können wir dieses lösen und erhalten die Lösung $a = \frac{1}{3}$, $b = \frac{1}{2}$, $c = \frac{1}{6}$, $d = 0$, also vermuten wir

$$S'_n \stackrel{!}{=} \frac{n(n+1)(2n+1)}{6}. \tag{11}$$

3) Beweise Gleichung (11) mit vollständiger Induktion über n.

Induktionsanfang ($n = 1$): Es ist $S'_1 = 1 = \frac{1 \cdot 2 \cdot 3}{6} = \frac{1(1+1)(2 \cdot 1+1)}{6}$, also ist die Behauptung für $n = 1$ wahr.

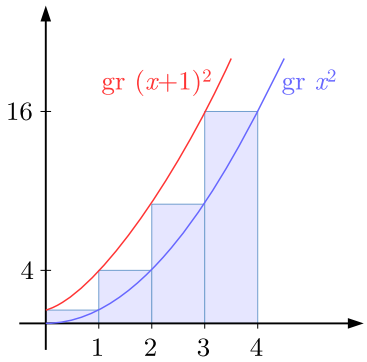


Abb. 1.9 Abschätzung von $\sum_{k=1}^{n} k^2$ nach unten (blau) und nach oben (rot) durch $\int_0^n x^2 \,\mathrm{dx}$ bzw. $\int_0^n (x+1)^2 \,\mathrm{dx}$.

Induktionshypothese: Angenommen, es gelte $S'_n = \frac{n(n+1)(2n+1)}{6}$ für ein $n \in \mathbb{N}$.
Induktionsschritt $(n \to n+1)$:

$$\begin{aligned} S'_{n+1} &= \sum_{k=1}^{n+1} k^2 = \Big(\sum_{k=1}^{n} k^2\Big) + (n+1)^2 = S'_n + (n+1)^2 \\ &\overset{I.H.}{=} \frac{n(n+1)(2n+1)}{6} + \frac{6(n+1)^2}{6} \\ &= \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)\cdot 2}{6} + \frac{2(n+1)(2n+3)}{6} \\ &= \frac{n(n+1)(2n+3)}{6} + \frac{2(n+1)(2n+3)}{6} \\ &= \frac{(n+2)(n+1)(2n+3)}{6} \\ &= \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}. \end{aligned}$$

∎

Abschätzung einer Summe durch Integrale

Idee 2 Ein Integral hat die anschauliche Interpretation, den Flächeninhalt unter dem Graphen im entsprechenden Intervall anzugeben. Auch die Summe $\sum_{k=1}^{n} k^2$ können wir geometrisch interpretieren, nämlich als Summe der Flächeninhalte von Rechtecken mit der Breite 1 und der Höhe k^2 (für $k = 1, 2, \ldots, n$). Abbildung 1.9 zeigt die Situation. Es sollte nun möglich sein, die Summe durch geeignete Integrale nach oben und nach unten abzuschätzen. Die Hoffnung besteht darin, dass diese Integrale wesentlich einfacher auszurechnen sind.

Tatsächlich gilt, wie Abbildung 1.9 zeigt, die Abschätzung

$$\int_0^n x^2 \,\mathrm{dx} \le \sum_{k=1}^{n} k^2 \le \int_0^n (x+1)^2 \,\mathrm{dx}. \tag{12}$$

Lösen wir diese Integrale nun, dann erhalten wir

$$\frac{1}{3}n^3 \le \sum_{k=1}^{n} k^2 \le \frac{1}{3}(n+1)^3 = \frac{1}{3}n^3 + n^2 + n + \frac{1}{3}.$$

Somit erhalten wir zwar keine exakte Abschätzung, wir sehen aber, dass $\sum_{k=1}^{n} k^2 = \frac{1}{3}n^3 + \mathcal{O}(n^2) \subseteq \Theta(n^3)$ gilt. Wir können die Idee zu folgender Aussage verallgemeinern: Sei $r \in \mathbb{N}_0$ beliebig. Für alle $n \in \mathbb{N}$ gilt

$\sum_{k=1}^{n} k^r$

$$\sum_{k=1}^{n} k^r = \frac{1}{r+1}n^{r+1} + \mathcal{O}(n^r). \tag{13}$$

Eine weitere sehr wichtige Summenart, die uns immer wieder begegnen wird, ist die sog. *geometrische Summe*. Sie hat die Form $\sum_{k=0}^{n} x^k$, und kann zu

GEOMETRISCHE SUMME

$\sum_{k=1}^{n} x^k$

$$S_n'' = \sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \tag{14}$$

aufgelöst werden. Diese Auflösung kann man leicht mit einem ähnlichen Beweis, wie wir ihn bereits vorhin zur Auflösung von $\sum_{k=1}^{n} k$ benutzt haben, verwendet werden. Wir schreiben wieder S_n'' hin, und darunter xS_n''. Es ergibt sich

$$\begin{array}{rcccccccccccc} S_n'' & = & 1 & + & x & + & x^2 & + & \cdots & + & x^n & & \\ xS_n'' & = & & & x & + & x^2 & + & \cdots & + & x^n & + & x^{n+1} \\ \hline S_n'' - xS_n'' & = & 1 & & & & & & & & & - & x^{n+1}, \end{array}$$

was die Auflösung (14) beweist. Sie bedeutet auch, dass

$$\sum_{k=0}^{\infty} x^k = \lim_{n\to\infty} \sum_{k=0}^{n} x^k = \frac{1}{1 - x} \text{ für } |x| < 1 \tag{15}$$

gilt. Eine letzte Summe wollen wir untersuchen, und zwar $\sum_{k=0}^{n} k \cdot x^k$. Auch hier ist nicht offensichtlich, wie sie aufgelöst werden kann, daher braucht es wieder einen Trick. Wir setzen $f(x) = \sum_{k=0}^{n} x^k$ und beobachten, dass $f'(x) = \sum_{k=0}^{n} k \cdot x^{k-1}$ gilt. Daher ergibt sich $\sum_{k=0}^{n} k \cdot x^k = x \cdot f'(x)$. Warum hat uns das geholfen? Wir haben ja bereits vorher eine explizite Formel für $f(x)$ bewiesen, nämlich $\frac{1-x^{n+1}}{1-x}$. Also erhalten wir $f'(x)$, indem wir diesen Ausdruck einfach differenzen. Wir haben also die Berechnung einer Summe auf die Berechnung einer anderen Summe zurückgeführt.

$\sum_{k=0}^{n} k \cdot x^k$

1.4.3 Kombinatorik

Permutationen Im Folgenden sei M eine Menge mit $n < \infty$ Elementen. Eine bijektive Abbildung $\pi : M \to M$ heisst *Permutation*. Solch eine Permutation kann man als Anordnung $(\pi(1), \pi(2), \ldots, \pi(n))$ der Menge M verstehen: An erster Stelle steht das Element $\pi(1)$, an zweiter Stelle $\pi(2)$, usw. So zum Beispiel hat die Menge $M = \{1, 2, 3\}$ genau 6 Permutationen, nämlich $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ und $(3, 2, 1)$. Mengen mit n Elementen haben

PERMUTATION

$$n \cdot (n - 1) \cdots 2 \cdot 1 =: n! \tag{16}$$

(sprich: n *Fakultät*) viele Permutationen, denn für das Element $\pi(1)$ gibt es n Möglichkeiten, für $\pi(2)$ noch $n - 1$ (alle Elemente in M ausser $\pi(1)$), usw.

FAKULTÄT

Wir wollen nun untensuchen, wie schnell $n!$ wächst. Es ist leicht ersichtlich, dass $2^n \leq n! \leq n^n$ für $n \geq 2$ gilt. Um eine bessere Abschätzung zu erhalten, beobachten wir zunächst, dass $\ln(a \cdot b) = \ln(a) + \ln(b)$ und damit auch $\ln(n!) = \sum_{i=1}^{n} \ln(i)$ gilt. Diese Summe können wir nun wie im vorigen Abschnitt gesehen durch Integrale geeignet nach unten und nach oben abschätzen, und erhalten

WACHSTUM VON $n!$

$$\ln(n!) = n \ln(n) - n + \mathcal{O}(\ln n) \tag{17}$$

$$\Rightarrow n! = e^{\mathcal{O}(\ln n)} \cdot \left(\frac{n}{e}\right)^n. \tag{18}$$

Man beachte, dass $e^{\mathcal{O}(\ln n)} \neq \mathcal{O}(n)$ ist, denn $e^{\mathcal{O}(\ln n)}$ enthält zum Beispiel die Funktion $e^{2 \ln n} = (e^{\ln n})^2 = n^2$, die nicht in $\mathcal{O}(n)$ liegt. Eine genauere Abschätzung liefert die *Stirling-Formel*

STIRLING-FORMEL

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{19}$$

Potenz- und Teilmengen Die Menge aller Teilmengen von M, $\mathcal{P}(M) = \{A \mid A \subseteq M\}$, heisst *Potenzmenge von* M. Hat $|M| = n$ Elemente, dann hat die Potenzmenge von M Kardinalität $|\mathcal{P}(M)| = 2^n$. Um dies zu sehen, nummerieren wir die Elemente von M zunächst von 1 bis n. Jede Teilmenge M' von M kann nun durch einen Bitvektor $(b_1, \ldots, b_n)$ der Länge n codiert werden, wobei b_i genau dann 1 ist, wenn $i \in M'$ gilt, und 0 sonst. Da es 2^n viele Bitvektoren der Länge n gibt und jeder Bitvektor zu genau einer Teilmenge von M korrespondiert, hat die Potenzmenge von M folglich 2^n viele Elemente.

POTENZMENGE

ANZAHL k-ELEMENTIGER TEILMENGEN

Es gibt also 2^n Teilmengen von M. Wie viele Teilmengen der Grösse k gibt es? Für das erste Element haben wir n Möglichkeiten, für das zweite $n-1$, und für das k-te $n-k+1$. Insgesamt ergeben sich so $n \cdot (n-1) \cdots (n-k+1)$ viele Möglichkeiten. Allerdings haben wir einige Teilmengen doppelt gezählt: Da die Reihenfolge der Elemente in der Teilmenge selbst egal ist und jede k-elementige Menge auf $k!$ viele Arten angeordnet werden kann, wurde in obigem Produkt jede Menge $k!$ Mal gezählt. Folglich müssen wir noch durch $k!$ teilen. Die korrekte Anzahl k-elementiger Teilmengen einer n-elementigen Menge ist also

$$\frac{n \cdot (n-1) \cdots (n-k+1)}{k!} = \frac{n!}{k!(n-k)!} =: \binom{n}{k}. \tag{20}$$

BINOMIAL-KOEFFIZIENT

Den Ausdruck $\binom{n}{k}$ bezeichnen wir als *Binomialkoeffizient* und sagen dazu "n über k" (engl.: "n *choose* k"). Er ist für alle $n, k \in \mathbb{N}_0$ mit $0 \leq k \leq n$ definiert. Es stellt sich aber die Frage, was $\binom{n}{0}$ ist, denn bisher haben wir 0! noch nicht definiert. Wir beobachten, dass es genau eine 0-elementige Teilmenge von M gibt (nämlich die leere Menge), folglich sollte $\binom{n}{0} = 1$ sein. Damit erhalten wir $\binom{n}{0} = \frac{n!}{0!(n-0)!} = \frac{1}{0!} = 1$, und aus diesem Grund definieren wir $0! := 1$. Für Binomialkoeffizienten können wir viele Identitäten beweisen, zum Beispiel die Folgenden:

IDENTITÄTEN

- $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. Die Aussage können wir mit einem *kombinatorischen Beweis* zeigen, indem wir geeignete kombinatorische Objekte (hier: Teilmengen von M) auf zwei verschiedene Arten zählen. Zum einen beobachten wir, dass es genau $\binom{n}{k}$ k-elementige Teilmengen von M gibt (linke Seite). Zum anderen haben wir aber auch gesehen, dass M insgesamt 2^n viele Teilmengen besitzt (rechte Seite).

- $\binom{n}{n-k} = \binom{n}{k}$, denn für jede k-elementige Teilmenge M' können wir auch die $(n-k)$-elementige Teilmenge $M'' = M \backslash M'$ zählen. Alternativ kann man die Aussage natürlich auch beweisen, indem man die Definition von $\binom{n}{n-k}$ hinschreibt und durch geeignete Termumformung in $\binom{n}{k}$ überführt.

- $\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$ für $1 \leq k \leq n$. Dies kann ebenfalls durch Termumformung der Definition bewiesen werden:

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-1-k)!} \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!} \cdot \Big(\frac{1}{n-k} + \frac{1}{k}\Big) \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!} \cdot \Big(\frac{n}{k(n-k)}\Big) = \binom{n}{k}.\end{aligned}$$

- $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ für alle $x, y \in \mathbb{R}$. Dies kann durch vollständige Induktion über n bewiesen werden (Übung).

1.4.4 Rekurrenzen

Wir haben bereits früher Rekursionsgleichungen kennengelernt, nämlich

$$F(n) = F(n-1) + 2(n-1) \quad \text{(Typ 1, Star finden), sowie} \tag{21}$$

$$M(n) = 3 \cdot M\Big(\frac{n}{2}\Big) \quad \text{(Typ 2, Algorithmus von Karatsuba und Ofman).} \tag{22}$$

Wir betrachten und lösen diese Gleichungen nun in einem etwas allgemeineren Kontext. Sei $C : \mathbb{N}_0 \to \mathbb{R}$ eine beliebige Funktion. Gleichungen vom Typ 1 können wir nun zu

$$S(0) = C(0), \quad S(n) = a \cdot S(n-1) + C(n) \text{ für } n \geq 1 \tag{23}$$ Typ 1

verallgemeinern, wobei wir annehmen wollen, dass $a \in \mathbb{N}$ konstant ist. Gleichungen dieses Typs werden immer dann verwendet, wenn ein Algorithmus ein Problem der Grösse n in a Probleme der Grösse $n-1$ zerlegt, und zusätzlich $C(n)$ zusätzliche Operationen ausführt. Die Rekursionsgleichung (23) hat die Lösung

$$S(n) = \sum_{i=0}^{n} a^i C(n-i) = a^n C(0) + \sum_{i=0}^{n-1} a^i C(n-i), \tag{24}$$

was sich leicht mit vollständiger Induktion über n beweisen lässt (Übung). Gleichungen vom Typ 2 verallgemeinern wir zu

$$T(1) = D(1), \quad T(n) = a \cdot T\Big(\frac{n}{2}\Big) + D(n) \text{ für } n = 2^k,\ k \geq 1, \tag{25}$$ Typ 2

wobei wir wie vorher annehmen, dass $a \in \mathbb{N}$ konstant und $D : \mathbb{N} \to \mathbb{R}$ eine beliebige Funktion ist. Gleichungen vom Typ 2 werden in der Analyse von Algorithmen verwendet, die ein Problem der Grösse n in a Teilprobleme der Grösse $n/2$ zerlegen und zusätzlich $D(n)$ zusätzliche Operationen ausführen. Gleichungen vom Typ 2 können gelöst werden, indem Sie in eine Gleichung vom Typ 1 überführt und analog zu (24) aufgelöst werden (die Details auszuarbeiten ist Teil einer Übung).

1.5 Maximum Subarray Sum

Viele Algorithmen operieren "induktiv", d.h. sie lösen zuerst rekursiv ein oder mehrere Teilprobleme kleinerer Grösse und benutzen ihre Lösungen, um daraus die endgültige Lösung zu konstruieren. Sowohl der Algorithmus zum Finden eines Stars (1 Teilproblem mit Grösse $n-1$) als auch der Algorithmus von Karatsuba und Ofman (3 Teilprobleme mit Grösse $n/2$) arbeiten nach diesem Prinzip. Im Allgemeinen gibt es für ein Problem mehrere Algorithmen, die dieses Problem lösen. Unser Ziel besteht jeweils in der Entwicklung des (asymptotisch) effizientesten. Dabei interessieren uns insbesondere die Kosten (also die Anzahl durchgeführter elementarer Operationen) sowie der Speicherbedarf. Wir wollen in diesem Abschnitt noch einmal den Prozess des systematischen Algorithmenentwurfs anschauen und betrachten daher das *Maximum Subarray Sum Problem*. In diesem ist ein Array[1] von n rationalen Zahlen $a_1, \ldots, a_n$ gegeben, und gesucht ist ein Teilstück mit maximaler Summe (d.h., Problemdefinition

[1] Wir werden bald genau lernen, was ein Array ist. Im Moment betrachten wir es als Menge von Elementen in einer gegebenen Reihenfolge.

Indizes i, j mit $1 \leq i \leq j \leq n$, sodass $\sum_{k=i}^{j} a_k$ maximal ist). Zusätzlich erlauben wir auch die leere Summe als Lösung, sodass der Algorithmus immer eine Lösung mit nicht-negativem Wert zurückliefert. Wäre die Eingabe zum Beispiel

$$(a_1, \ldots, a_9) = (7, -11, 15, 110, -23, -3, 127, -12, 1),$$

dann wären die gesuchten Indizes $i = 3$ und $j = 7$ mit $\sum_{k=i}^{j} a_k = 226$.

Vereinfachende Annahmen

Bei manchen Eingaben gibt es mehr als eine maximale Lösung. Wir begnügen uns aber mit der Berechnung *irgendeiner* dieser maximalen Lösungen (welcher, ist egal). Ebenso berechnen die im folgenden vorgestellten Algorithmen nur den *Wert* einer besten Lösung (also z.B. 226 für die obige Eingabe) und nicht die Indizes selbst. Die Algorithmen könnten aber sehr einfach modifiziert werden, sodass nicht allein der beste Wert, sondern auch die zugehörigen Indizes selbst gespeichert würden.

Das Maximum Subarray Sum Problem hat zum Beispiel die folgende anschauliche Interpretation. Wenn die Zahlen Änderungen eines Aktienkurses beschreiben, dann möchte man rückwirkend berechnen, wann der beste Zeitpunkt für den Kauf (i) bzw. den Verkauf (j) gewesen wäre.

Algorithmus 1 (Naiv) Eine einfache Idee besteht darin, einfach alle möglichen Intervalle auszuprobieren, die Summe S im entsprechenden Intervall auszurechnen und sich die maximale Summe zu merken. Wir erhalten also den folgenden Algorithmus:

Naiver Algorithmus

MSS-Naiv$(a_1, \ldots, a_n)$

```
1 maxS ← 0
2 Für i ← 1, ..., n (alle Anfänge)
3     Für j ← i, ..., n (alle Enden)
4         S ← Σ_{k=i}^{j} a_k (berechne Summe)
5         Merke maximales S
```

Analyse

Theorem 1.2. *Der naive Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n^3)$ viele Additionen durch.*

Beweis. Wir beobachten, dass die Berechnung der Summe in Schritt 3 genau $j - i$ viele einzelne Additionen benötigt. Damit beträgt die Anzahl insgesamt durchgeführter Additionen genau

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) &= \sum_{i=1}^{n}\sum_{j'=0}^{n-i} j' = \sum_{i=1}^{n}\sum_{j'=1}^{n-i} j' \\
&= \sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2} \\
&= \sum_{i'=0}^{n-1} \frac{i'(i'+1)}{2} = \frac{1}{2}\Big(\sum_{i'=1}^{n-1}(i')^2 + \sum_{i'=1}^{n-1} i'\Big) \\
&= \frac{1}{2}\big(\Theta(n^3) + \Theta(n^2)\big) = \Theta(n^3). \quad \blacksquare
\end{aligned}$$

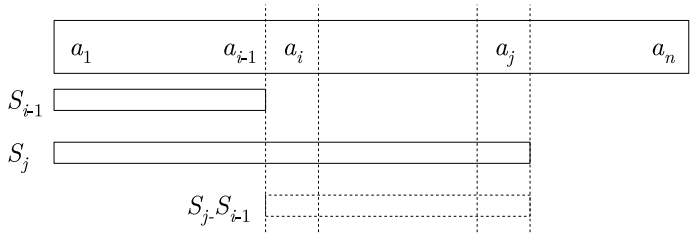


Abb. 1.10 Zur Berechnung von $\sum_{k=i}^{j} a_k$ genügt es, S_{i-1} von S_j zu subtrahieren.

Weitere Operationen

Man könnte argumentieren, dass nicht allein die Berechnung der Summe, sondern auch andere Schritte wie zum Beispiel das Merken des Maximums gezählt werden sollten. Eine genauere Analyse zeigt aber, dass die Anzahl aller Merke-Operationen durch $\mathcal{O}(n^2)$ nach oben beschränkt ist, was an der Gesamtanzahl von $\Theta(n^3)$ vielen Operationen nichts ändert.

Java-Implementierung

Der eben vorgestellte Algorithmus ist der Einfachheit halber in Pseudocode formuliert worden, damit die zugrundeliegende Idee direkt ersichtlich ist. In einer konkreten Programmiersprache gibt es viel mehr technische Details, die man beachten muss. In Java zum Beispiel könnte ein entsprechendes Codefragment wie folgt aussehen (man beachte, dass in den anderen Vorlesungen noch nicht alle Konzepte vorgestellt wurden):

```
double maximalesS=0.0;
for (int i=0; i<n; i=i+1)                    // Anfang
    for (int j=i; j<n; j=j+1) {              // Ende
        double S=0.0;
        for (int k=i; k<=j; k=k+1) {         // Berechne Summe
            S=S+a[k];
            if (S>maximalesS)                // Aktualisiere Maximum
                maximalesS=S;
        }
    }
```

Algorithmus 2 (Vorberechnung der Präfixsummen) Ein Problem des naiven Algorithmus ist, dass manche Teilsummen wieder und wieder neu berechnet werden, obwohl sie bereits früher aufsummiert wurden. Offenbar genügt es auch, lediglich alle *Präfixsummen* zu berechnen: Wir berechnen für jede Position i die Summe $S_i = \sum_{k=1}^{i} a_i$, also Summe der Zahlen von Position 1 bis und mit Position i. All diese Werte S_i können wir für aufsteigende i in linearer Zeit, also in $\mathcal{O}(n)$, berechnen. Zur Berechnung von S_i genügt es nämlich, die i-te Zahl zur bereits berechneten Summe S_{i-1} zu addieren. Wie können wir diese Präfixsummen nun benutzen? Wir beobachten, dass

$$\sum_{k=i}^{j} a_k = \Big(\sum_{k=1}^{j} a_k\Big) - \Big(\sum_{k=1}^{i-1} a_k\Big) = S_j - S_{i-1} \tag{26}$$

gilt. Zur Berechnung der Summe aller Zahlen im Bereich $i, i+1, \ldots, j-1, j$ reicht es also aus, S_{i-1} von S_j abzuziehen (siehe Abbildung 1.10). Da sowohl S_{i-1} als auch S_j bereits vorberechnet wurden, kann $\sum_{k=i}^{j} a_k$ also in Zeit $\mathcal{O}(1)$ berechnet werden. Wir erhalten also den folgenden Algorithmus:

Präfixsummen-Algorithmus

MSS-Präfixsummen$(a_1, \ldots, a_n)$

```
1 S_0 ← 0
2 Für i ← 1, ..., n
3     S_i ← S_{i-1} + a_i
4 maxS ← 0
5 Für i ← 1, ..., n
6     Für j ← i, ..., n
7         S ← S_j − S_{i-1}
8         Merke maximales S
```

In den Schritten 1–3 werden also also $n+1$ Präfixsummen $S_0, \ldots, S_n$ vorberechnet. Die Schritte 4–7 benutzen dann diese Präfixsummen zur Berechnung der entsprechenden Teilsummen $\sum_{k=i}^{j} a_k$.

Analyse

Theorem 1.3. *Der Präfixsummen-Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n^2)$ viele Additionen und Subtraktionen durch.*

> **Beweis.** Die Berechnung der Präfixsummen $S_0, \ldots, S_n$ in den Schritten 1–3 erfordert genau n Additionen. In Schritt 6 wird genau eine Subtraktion durchgeführt, also ist die Gesamtanzahl durchgeführter Subtraktionen genau
>
> $$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \sum_{i=1}^{n} (n - i + 1) = \sum_{i'=1}^{n} i' = \Theta(n^2). \quad \blacksquare$$

Algorithmus 3 (Divide-and-Conquer) Wir haben bereits früher das Prinzip von Divide-and-Conquer kennengelernt, bei dem ein Problem zunächst in kleinere Teilprobleme zerlegt wird, aus deren Lösung dann die Lösung für das Gesamtproblem berechnet wird (wie zum Beispiel im Algorithmus von Karatsuba und Ofman). Wir wollen nun dieses Prinzip auch zur Lösung des Maximum Subarray Sum Problems einsetzen. Dazu teilen wir die gegebenen Zahlen $a_1, \ldots, a_n$ zunächst in zwei gleich (annähernd) grosse Hälften $\langle a_1, \ldots, a_{\lfloor n/2 \rfloor} \rangle$ und $\langle a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n \rangle$ auf. Der Einfachheit halber nehmen wir im Folgenden an, dass $n = 2^k$ für ein $k \in \mathbb{N}_0$ gilt, d.h. n eine Zweierpotenz ist. Sind nun i, j mit $1 \leq i \leq j \leq n$ die Indizes einer maximalen Lösung (d.h., $\sum_{k=i}^{j} a_k$ ist maximal), dann können exakt drei Fälle auftreten (siehe auch Abbildung 1.11):

1. Fall: Die Lösung liegt vollständig in der ersten Hälfte ($1 \leq i \leq j \leq n/2$).

2. Fall: Die Lösung liegt vollständig in der zweiten Hälfte ($n/2 < i \leq j \leq n$).

3. Fall: Die Lösung läuft über die Mitte ($1 \leq i \leq n/2 < j \leq n$).

Wir beobachten nun, dass die ersten beiden Fälle durch einen rekursiven Aufruf auf den ersten bzw. den letzten $n/2$ Elementen gelöst werden können. Diese rekursiven Aufrufe zerlegen ihrerseits wieder die Eingabe in zwei Hälften, bis diese nur noch ein Element enthalten.

Für den dritten Fall machen wir nun folgende entscheidende Beobachtung: Läuft die Lösung über die Mitte, dann ist sie aus genau zwei Teilen zusammengesetzt,

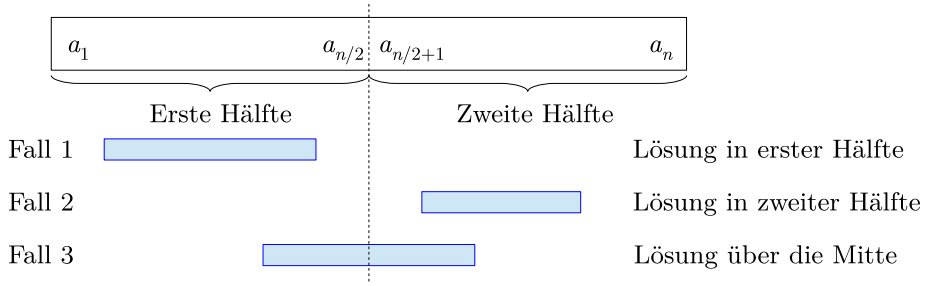


Abb. 1.11 Die drei möglichen Fälle, die beim Divide-and-Conquer-Algorithmus auftreten können.

nämlich dem besten Teilstück, das aus den letzten Elementen der ersten und den ersten Elementen der zweiten Hälfte besteht. Formaler gesprochen betrachten wir alle Suffixsummen $S_1, \ldots, S_{n/2}$ mit $S_i = \sum_{k=i}^{n/2} a_k$ (also die Summen der Elemente i bis und mit $n/2$) und alle Präfixsummen $P_{n/2+1}, \ldots, P_n$ mit $P_j = \sum_{k=n/2+1}^{j} a_k$ (also die Summen der Elemente $n/2+1$ bis und mit j), und die beste Lösung, die in beiden Hälften liegt, hat dann den Wert $\max_i S_i + \max_j P_j$. Um den Wert einer besten Lösung zu ermitteln, die über die Mitte läuft, reicht es also aus, die grösste Suffixsumme in der ersten und die grösste Präfixsumme in der zweiten Hälfte zu berechnen, und diese Werte dann zu addieren.

Natürlich wissen wir im Vorfeld nicht, ob die optimale Lösung komplett in einer der beiden Hälften liegt oder in beiden. Das macht aber nichts, denn wir können einfach rekursiv die beste Lösung in der ersten Hälfte, rekursiv die beste Lösung in der zweiten Hälfte sowie die beste Lösung über die Mitte wie oben beschrieben berechnen, und geben dann die beste dieser drei Lösungen (bzw. ihren Wert) zurück.

Divide-and-Conquer-Algorithmus

MSS-Divide-and-Conquer$(a_1, \ldots, a_n)$

1 Wenn $n = 1$ ist, dann gib $\max\{a_1, 0\}$ zurück.
2 Wenn $n > 1$ ist:
3 Teile die Eingabe in $A_1 = \langle a_1, \ldots, a_{n/2} \rangle$ und $A_2 = \langle a_{n/2+1}, \ldots, a_n \rangle$ auf.
4 Berechne rekursiv den Wert W_1 einer besten Lösung für das Array A_1.
5 Berechne rekursiv den Wert W_2 einer besten Lösung für das Array A_2.
6 Berechne grösste Suffixsumme S in A_1.
7 Berechne grösste Präfixsumme P in A_2.
8 Setze $W_3 \leftarrow S + P$.
9 Gib $\max\{W_1, W_2, W_3\}$ zurück.

Analyse

Theorem 1.4. *Der Divide-and-Conquer-Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n \log n)$ viele Additionen und Vergleiche durch.*

Beweis. Wir beobachten zunächst einmal, dass Schritt 1 nur konstante Kosten c hat. Schritte 3, 8 und 9 sind bereits in $\mathcal{O}(1)$. Die Schritte 6 und 7 haben Kosten $\mathcal{O}(n)$, denn es müssen $n/2$ Suffix- und $n/2$ Präfixsummen berechnet werden. Da S_i aus S_{i+1} und P_{j+1} aus P_j mittels einer Addition berechnet werden können, ist der Gesamtaufwand in $\mathcal{O}(n)$, d.h.,

er ist maximal $a \cdot n$ mit einer Konstante a. Beziehen wir die rekursiven Aufrufe mit ein, erhalten wir also folgende Rekurrenz zur Beschreibung der Kosten bei einer Eingabe mit n Zahlen:

$$T(n) = \begin{cases} c & \text{falls } n = 1 \text{ ist} \\ 2T\left(\frac{n}{2}\right) + an & \text{falls } n > 1 \text{ ist.} \end{cases} \tag{27}$$

Da nun $n = 2^k$ gilt, definieren wir $\bar{T}(k) = T(2^k) = T(n)$, d.h., aus Gleichung (27) wird

$$\bar{T}(k) = \begin{cases} c & \text{falls } k = 0 \text{ ist} \\ 2\bar{T}(k-1) + a \cdot 2^k & \text{falls } k > 0 \text{ ist.} \end{cases} \tag{28}$$

Wie früher gesehen hat Gleichung (28) die Auflösung

$$\bar{T}(k) = 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} = c \cdot 2^k + a \cdot k \cdot 2^k = \Theta(k \cdot 2^k), \tag{29}$$

und damit ist $T(n) = \Theta(n \log n)$. Man könnte sich nun fragen, warum wir die Konstanten c und a nicht genauer bestimmt haben. Der Grund ist, dass der genaue Wert keine Rolle spielt, da diese beiden Konstanten in der Lösung von $\bar{T}(k)$ nur als konstante Faktoren auftauchen und daher in der Θ-Notation verschwinden. ▮

Algorithmus 4 (Induktiv von links nach rechts) Können wir es noch besser schaffen? Dazu versuchen wir nochmals eine Induktion von links nach rechts. Dann nehmen wir an, dass wir nach dem $(i-1)$-ten Schritt den Wert max einer optimalen Lösung für die ersten $i-1$ Elemente kennen (initial: max = 0). Wenn wir nun das i-te Element dazunehmen, dann wollen wir rasch herausfinden, ob dieses neue Element zum besten Intervall gehört. Dazu merken wir uns für den Teil bis i nicht nur das bisherige Maximum, sondern auch den Wert randmax einer besten Lösung, die am rechten Rand, also an Position i, endet (initial: randmax = 0). Damit können wir dann leicht prüfen, ob wir mit dem neuen Element ein neues Maximum erreichen können oder nicht (siehe Abbildung 1.12).

Induktiver Algorithmus

MSS-Induktiv$(a_1, \ldots, a_n)$

```
1 randmax ← 0
2 maxS ← 0
3 Für i ← 1, ..., n:
4     randmax ← randmax + a_i
5     Wenn randmax > maxS:
6         maxS ← randmax
7     Wenn randmax < 0:
8         randmax ← 0
9 Gib maxS zurück.
```

Analyse

Theorem 1.5. *Der induktive Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n)$ viele Additionen und Vergleiche durch.*

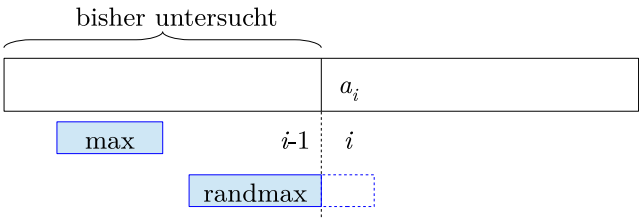


Abb. 1.12 Ein grösseres Maximum kann nur durch randmax + a_i entstehen.

Komplexität des Problems Nun werden wir ehrgeizig und wollen es noch besser machen. Können wir auch Algorithmen mit Kosten von $\Theta(\sqrt{n})$ oder sogar $\Theta(\log(n))$ entwerfen, oder ist geht es nicht besser als $\Theta(n)$?

Wir überlegen uns nun, dass jeder korrekte Algorithmus für das Maximum Subarray Sum Problem Kosten $\Omega(n)$ hat, indem wir zeigen, dass jeder korrekte Algorithmus *alle* Elemente $a_1, \ldots, a_n$ ansehen muss.

Angenommen, wir haben einen Algorithmus, der nicht jedes Element anschauen muss (das wäre etwa dann der Fall, wenn wir einen Algorithmus mit Laufzeit $\Theta(\sqrt{n})$ hätten). Dann ist er nicht korrekt:

UNTERE SCHRANKE

Fall 1: Der Algorithmus berechnet eine Lösung, die a_i enthält. Dann setzen wir $a_i := -\infty$ (oder auf einen Wert, der so klein ist, dass a_i auf keinen Fall Teil einer optimalen Lösung ist, z.B. $-n \max_k |a_k| - 1$).

Fall 2: Der Algorithmus berechnet eine Lösung, die a_i nicht enthält. Dann setzen wir $a_i := \infty$ (oder auf einen Wert, der so gross ist, dass a_i auf jeden Fall Teil einer optimalen Lösung sein muss, z.B. $n \max_k |a_k| + 1$).

In beiden Fällen berechnet ein Algorithmus, der a_i nicht anschaut, eine falsche Lösung. Damit sind wir in der glücklichen (und seltenen) Lage, die Komplexität des Maximum Subarray Sums exakt einschätzen zu können. Sie ist in

PROBLEMKOMPLEXITÄT

$$\Omega(n) \cap \mathcal{O}(n) = \Theta(n). \tag{30}$$

Die untere Schranke von $\Omega(n)$ erhalten wir durch den soeben geführten Beweis, die obere Schranke von $\mathcal{O}(n)$ durch den induktiven Algorithmus.

Kapitel 2

Sortieren und Suchen

2.1 Suchen in sortierten Arrays

Angenommen, wir haben ein Telefonbuch mit einer Million Einträge, und wir wollen einen bestimmen Namen finden. Formaler nehmen wir an, dass ein *sortiertes* Array A mit n Elementen (*Schlüsseln*) $A[1], \ldots, A[n]$ und $A[1] \leq A[2] \leq \cdots \leq A[n]$ sowie ein Element (Schlüssel) b gegeben sind, und gesucht ist ein Index k mit $1 \leq k \leq n$, sodass $A[k] = b$ ist, bzw. "nicht gefunden" falls ein solcher Index nicht existiert. Die Aufgabe besteht also darin zu entscheiden, ob A den Schlüssel b enthält, und wenn ja, die entsprechende Position im Array zu berechnen.

2.1.1 Binäre Suche

Idee

Eine naheliegende Strategie besteht darin, das Suchproblem mittels Divide-and-Conquer zu lösen. Dazu könnte man wie folgt vorgehen: Ist das Array leer, dann enthält es b mit Sicherheit nicht und wir geben "nicht gefunden" aus. Ansonsten bestimmen wir die (annähernd) mittlere Position $m = \lfloor n/2 \rfloor$ im Array und unterscheiden drei Fälle:

1) Ist $b = A[m]$, dann haben wir den Schlüssel b an Position m gefunden und geben daher m zurück.

2) Ist $b < A[m]$, dann suchen wir rekursiv in der linken Hälfte (also auf den Positionen $1, \ldots, m-1$) weiter nach b.

3) Ist $b > A[m]$, dann suchen wir rekursiv in der rechten Hälfte (also auf den Positionen $m+1, \ldots, n$) weiter nach b.

Korrektheit

Dieses Verfahren wird *binäre Suche* genannt. Die Korrektheit der binären Suche folgt direkt aus der Tatsache, dass A aufsteigend sortiert ist. Ist also $b < A[m]$, dann wissen wir bereits, dass die Positionen $m, \ldots, n$ nur Schlüssel enthalten, die grösser als b sind. Also reicht es, die Suche auf die ersten $m-1$ Positionen einzuschränken. Die Begründung für den Fall $b > A[m]$ verläuft analog. Da der Suchbereich (die Anzahl der noch möglichen Positionen) in jedem Schritt um mindestens ein Element kleiner wird, endet das Verfahren auch nach einer endlichen Zahl von Schritten.

Laufzeit

Wie viele Schritte sind dies im schlimmsten Fall? Dieser tritt offenbar dann ein, wenn erfolglos nach einem Schlüssel gesucht wird. Für ein Array der Länge $n = 2^k$ ergibt sich die folgende Rekurrenz zur Abschätzung der maximal durchgeführten Operationen bei einem Array A der Länge n:

$$T(n) = \begin{cases} c & \text{falls } n = 0 \text{ ist,} \\ T(n/2) + d & \text{falls } n \geq 1 \text{ ist,} \end{cases} \tag{31}$$

wobei c und d jeweils konstant sind. Mit den früher in der Vorlesung vorgestellten Methoden erhalten wir $T(n) \in \mathcal{O}(\log n)$, was sehr schnell ist. In der Realität würde man zudem die binäre Suche iterativ statt rekursiv implementieren, was nicht schwierig ist, wie der folgende Pseudocode zeigt.

Binäre Suche

BINARY-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 left ← 1; right ← n                                  ▷ Initialer Suchbereich
2 while left ≤ right do
3     middle ← ⌊(left+right)/2⌋
4     if A[middle] = b then return middle              ▷ Element gefunden
5     else if A[middle] > b then right ← middle−1      ▷ Suche links weiter
6     else left ← middle+1                             ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

2.1.2 Interpolationssuche

Zu Beginn dieses Abschnitts wurde die binäre Suche mit der Suche nach einem Namen in einem Telefonbuch verglichen, was nicht ganz angemessen ist: Namen wie "Becker" würde man direkt eher im vorderen Bereich suchen, während man "Wawrinka" vermutlich ziemlich weit hinten vermuten würde. Binäre Suche setzt aber in Schritt 4 immer

$$\text{middle} = \left\lfloor \text{left} + \frac{1}{2}(\text{right} - \text{left}) \right\rfloor \tag{32}$$

und teilt daher den Suchbereich unabhängig vom zu suchenden Element b in zwei gleich grosse Teile auf. Vermutet man b eher links im Array, dann sollte der Faktor $1/2$ verringert werden, um den linken Suchbereich zu verkleinern. Analog sollte der Faktor eher grösser als $1/2$ gewählt werden, wenn sich b vermutlich weit rechts im Array findet. Für Arrays aus Zahlen wird diese Idee von der *Interpolationssuche* aufgegriffen. Sie ersetzt den statischen Faktor $1/2$ durch

Erwartete Position des Suchschlüssels

$$\rho = \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \in [0, 1], \tag{33}$$

und betrachtet die Position left $+ \rho(\text{right} - \text{left})$, was annähernd der erwarteten Position des Suchschlüssels b im Array A entspricht, wenn die Schlüsselwerte in dem Bereich einigermassen gleich verteilt sind. Man kann zeigen, dass Interpolationssuche auf Arrays, die aus n unabhängigen und gleichverteilten Zufallszahlen bestehen, nur $\mathcal{O}(\log \log n)$ viele Vergleiche benötigt. Andererseits werden im schlimmsten Fall $\Omega(n)$ viele Vergleiche ausgeführt, was wesentlich schlechter als etwa die binäre Suche ist.

Laufzeit

2.1.3 Exponentielle Suche

Angenommen, wir haben starke Anzeichen dafür, dass sich der zu suchende Schlüssel weit vorn im Array befindet. Können wir etwas schneller als die herkömmliche binäre Suche sein, ohne eine lineare Laufzeit im schlimmsten Fall (wie bei der Interpolationssuche) zu riskieren?

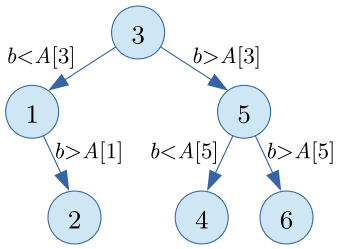


Abb. 2.1 Binäre Suche auf einem Array der Länge 6 als Entscheidungsbaum visualisiert. Da sich das zu suchende Element im Erfolgsfall auf sechs möglichen Positionen befinden kann, hat der Entscheidungsbaum die gleiche Anzahl von Knoten. Im ersten Schritt wird $A[3]$ mit b verglichen. Bei Gleichheit ist der Algorithmus fertig, ansonsten werden $A[1]$ bzw. $A[5]$ verglichen, usw. Da der Baum Höhe 3 hat (also aus drei "Schichten" besteht), ist die binäre Suche auf einem Array mit sechs Elementen nach spätestens drei Vergleichen fertig.

Um das Problem zu lösen, legen wir zunächst die rechte Grenze des Suchbereichs auf $r = 1$ fest. Diese wird dann so lange verdoppelt, bis entweder der dort gespeicherte Schlüssel $A[r]$ grösser als der zu suchende Schlüssel b ist, oder aber $r > n$ gilt (die rechte Grenze also ausserhalb des ursprünglichen Arrays liegt). Nachdem wir r auf diese Art bestimmt haben, führen wir eine binäre Suche auf dem Bereich $1, \ldots, \min(r, n)$ durch.

EXPONENTIELLE SUCHE

EXPONENTIAL-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 r ← 1                                          ▷ Initiale rechte Grenze
2 while r ≤ n and b > A[r]                       ▷ Finde rechte Grenze
3     r ← 2 · r                                  ▷ Verdopple rechte Grenze
4 return BINARY-SEARCH((A_1, ..., A[min(r, n)]), b)  ▷ Weiter mit binärer Suche
```

LAUFZEIT

Befindet sich das zu suchende Element auf Position p, dann braucht diese sog. *exponentielle Suche* nur $\mathcal{O}(\log p)$ viele Schritte (die Schleife im zweiten Schritt wird lediglich $\lceil \log p \rceil$ Mal durchlaufen, und die binäre Suche in Schritt 4 benötigt auch nur $\mathcal{O}(\log p)$ viele Schritte). Natürlich hat diese Art zu suchen nur dann Vorteile, wenn $p \ll n$ gilt. An der Laufzeit von $\mathcal{O}(\log n)$ im schlechtesten Fall ändert sich nichts.

2.1.4 Untere Schranke

Wie wir in den vorigen Abschnitten gesehen haben, benötigen sowohl die binäre als auch die exponentielle Suche im schlimmsten Fall $\Theta(\log n)$ viele Vergleiche; die Interpolationssuche kann sogar linear viele Vergleiche benötigen. Dies wirft die Frage auf, ob die Suche in einem sortierten Array im schlechtesten Fall vielleicht immer Zeit $\Omega(\log n)$ benötigt. Wir werden jetzt zeigen, dass alle Suchalgorithmen, die nur Vergleiche benutzen, im schlechtesten Fall tatsächlich mindestens so viele Vergleiche ausführen müssen.

ENTSCHEIDUNGSBAUM

Um zu argumentieren, dass es nicht schneller geht, stellen wir uns einen *beliebigen* deterministischen Suchalgorithmus als *Entscheidungsbaum* vor, der in jedem *Knoten* einen Schlüsselvergleich durchführt und dann je nach Ergebnis entweder erfolgreich endet oder in einem von zwei *Teilbäumen* fortfährt. Abbildung 2.1 visualisiert die binäre Suche auf einem Array mit sechs Elementen.

Wir beobachten nun folgendes: Ist eine Suche nach einem Element b erfolgreich, dann kann b an *jeder* der n Positionen des Arrays stehen. Für jede dieser n möglichen Ergebnisse der Suche muss der Entscheidungsbaum mindestens einen Knoten enthalten. Also muss die Gesamtanzahl der Knoten mindestens n betragen. Die Anzahl von Vergleichen, die ein Algorithmus im schlechtesten Fall ausführt, entspricht exakt der *Höhe* des Baums. Diese ist definiert als die maximale Anzahl von Knoten auf einem Weg von der *Wurzel* (dem "obersten" Knoten) zu einem *Blatt* (einem Knoten ohne Nachfolger). Wir beobachten nun, dass jeder Knoten maximal zwei Nachfolger hat, denn wir suchen entweder im linken Teilbaum weiter, oder im rechten (oder aber b wurde gefunden, dann ist die Suche beendet). Ein solcher Baum der Höhe h hat nun höchstens

$$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1 < 2^h \tag{34}$$

viele Knoten (denn die erste "Schicht" enthält nur die Wurzel, und die i-te "Schicht" höchstens doppelt so viele Knoten wie die darüberliegende $(i-1)$-te "Schicht"), und nun direkt folgt

$$n \leq \text{Anzahl Knoten im Entscheidungsbaum} < 2^h \Rightarrow h > \log_2(n). \tag{35}$$

Da die Anzahl der im schlimmsten Fall ausgeführten Vergleiche h beträgt und unsere Argumentation für *beliebige* Entscheidungsbäume gilt, haben wir also bewiesen dass jedes vergleichsbasierte Suchverfahren auf einem sortierten Array im schlechtesten Fall $\Omega(\log n)$ viele Vergleiche durchführt.

2.2 Suchen in unsortierten Arrays

Wir haben bereits gesehen, dass wir in sortierten Arrays in Zeit $\Theta(\log n)$ suchen können. Was aber, wenn das Array unsortiert ist? Die naivste Variante, in solchen Arrays zu suchen, ist die *lineare* Suche, bei der alle Elemente des Arrays A sukzessive mit dem suchenden Schlüssel b verglichen werden.

Lineare Suche

LINEAR-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 for i ← 1, 2, ..., n do
2     if A[i] = b then return i
3 return "nicht gefunden"
```

Laufzeit

Die Laufzeit der linearen Suche beträgt im schlechtesten Fall offensichtlich $\Theta(n)$. Tatsächlich werden wir jetzt zeigen, dass es nicht besser geht, denn *jeder* Algorithmus führt bei der Suche auf einem unsortierten Array im schlechtesten Fall n Vergleiche aus. Intuitiv könnte man argumentieren, dass der Schlüssel b ja mit allen Elementen in A vergleichen werden muss, sonst können wir nicht mit Sicherheit sagen, ob er in A vorkommt oder nicht. Das Argument ist aber nicht ausreichend, denn wir müssen auch Vergleiche von Schlüsseln innerhalb von A berücksichtigen. Könnten wir z.B. ein Array A in Zeit $\mathcal{O}(\log n)$ sortieren (was nicht möglich ist, wie wir später sehen werden), dann wäre eine Suche in Zeit $\mathcal{O}(\log n)$ möglich.

Abb. 2.2 Eine Situation, bei der $r = 5$ Vergleiche innerhalb von A durchgeführt wurden. Es gibt vier Gruppen, die in verschiedenen Farben dargestellt sind (Elemente der gleichen Farbe befinden sich in der gleichen Gruppe).

Untere Schranke Um nun zu beweisen, dass die Suche in ungeordneten Arrays im schlimmsten Fall immer $\Omega(n)$ viele Vergleiche ausführt, betrachten wir wieder einen *beliebigen* vergleichsbasierten Suchalgorithmus, und definieren r als die Anzahl durchgeführter Vergleiche von Schlüsseln innerhalb von A, und s als die Anzahl durchgeführter Vergleiche von Schlüsseln aus A mit b. Die lineare Suche z.B. vergleicht $r = 0$ Schlüssel innerhalb von A, und im schlimmsten Fall $s = n$ Schlüssel mit b (nämlich dann, wenn b das letzte Element ist, oder in A gar nicht vorkommt). Wir beobachten nun, dass die r Vergleiche von Schlüsseln innerhalb von A das Array A in g Gruppen unterteilen. Eine Gruppe enthält all diejenigen Schlüssel, die durch eine Kette von Vergleichen miteinander verbunden sind (vgl. Abbildung 2.2). Zu Beginn des Algorithmus ist kein Schlüssel aus A mit einem anderen Schlüssel aus A verglichen worden, folglich gibt es n Gruppen (jeder einzelne Schlüssel bildet eine Gruppe der Grösse 1). Um zwei Gruppen miteinander zu verbinden, reicht ein Vergleich. Da das Array n Schlüssel speichert (die anfangs n Gruppen darstellen) und wir annehmen, dass es g Gruppen gibt, waren dazu also mindestens $n - g$ viele Vergleiche nötig. Folglich gilt also auch $r \geq n - g$ (es müssen mindestens $n - g$ Schlüssel innerhalb von A verglichen werden). Auch können wir argumentieren, dass eine erfolglose Suche mindestens einen Vergleich pro Gruppe benötigt. Also müssen mindestens g Schlüssel mit b verglichen werden, und damit ist $s \geq g$. Insgesamt erhalten wir nun

Untere Schranke

$$r + s \geq (n - g) + g = n, \tag{36}$$

d.h., *jedes* Suchverfahren benötigt auf einem Array der Länge n im schlimmsten Fall mindestens n viele Vergleiche (entweder von Schlüsseln innerhalb von A, oder von Schlüsseln in A mit b).

2.3 Elementare Sortierverfahren

Im Zeitalter von Big Data ist es essentiell, Suchanfragen innerhalb kürzester Zeit beantworten zu können. Wir haben bereits gesehen, dass die Schlüssel in einem sortierten Array signifikant schneller als in einem unsortierten Array gefunden werden können. Deswegen wollen wir nun untersuchen, wie wir ein Array sortieren können. Formal definieren wir das Sortierproblem wie folgt:

Motivation

Eingabe: Ein Array $A = (A[1], \ldots, A[n])$ der Länge n

Ausgabe: Eine Permutation A' von A, die sortiert ist, d.h., es gilt $A'[i] \leq A'[j]$ für alle i, j mit $1 \leq i \leq j \leq n$.

Definition Sortierproblem

Oftmals wollen wir A' gar nicht getrennt von A berechnen, sondern wollen das Array A selbst so umordnen, dass es am Ende sortiert ist, d.h., die Ausgabe befindet sich an der Stelle, wo vorher die Eingabe stand. Sortieralgorithmen, die neben der Eingabe nur $\mathcal{O}(\log n)$ viel zusätzlichen Speicher benötigen, werden als "in-place"

oder "in-situ" bezeichnet. Wir werden im Folgenden verschiedene Sortieralgorithmen vorstellen und analysieren, wie viele Schlüsselvergleiche und wie viele Schlüsselbewegungen bzw. -vertauschungen sie jeweils im besten und im schlechtesten Fall ausführen. Weitere Kostenmasse wie z.B. der benutzte Speicher sind denkbar.

KOSTENMODELL

2.3.1 Bubblesort

SORTIERTHEIT PRÜFEN

Als Vorüberlegung für das erste vorgestellte Sortierverfahren machen wir uns zunächst klar, dass wir mit $n-1$ Vergleichen prüfen können, ob ein gegebenes Array A sortiert ist. Dazu prüfen wir, ob $A[1] \leq A[2]$ gilt, ob $A[2] \leq A[3]$ gilt, usw. Wenn mindestens einer dieser Vergleiche falsch ist, dann wissen wir sofort, dass das Array nicht sortiert ist.

SORTIERTHEIT PRÜFEN

IS-SORTED$(A = (A[1], \ldots, A[n]))$

```
1 for j ← 1 to n − 1 do
2     if A[i] > A[i + 1] then return "nicht sortiert"
3 return "sortiert"
```

MODIFIKATION

Wir versuchen nun zu sortieren, indem wir dieses Prüfverfahren ganz leicht modifizieren: Jedes Mal, wenn $A[i] > A[i+1]$ gilt, vertauschen wir die Schlüssel an den Positionen i und $i+1$ (denn sie befinden sich offenbar in der falschen Reihenfolge). Genügt dies, um ein sortiertes Array zu erhalten? Offenbar nicht, wie etwa das Array $(5, 9, 8, 7)$ zeigt: Nachdem das modifizierte Verfahren durchgelaufen ist, erhalten wir $(5, 8, 7, 9)$, was noch immer nicht sortiert ist. Wir machen aber eine entscheidende

BEOBACHTUNG

Beobachtung: Nach einem solchen Durchlauf ist das grösste Element an die letzte Position des Arrays gewandert. Wiederholen wir das Ganze, dann ist das zweitgrösste Element an die vorletzte Position verschoben worden, usw. Also sollten insgesamt $n-1$ Wiederholungen des obigen Verfahrens zur Sortierung des Arrays ausreichen. Da man sich anschaulich vorstellen kann, dass die Schlüssel im Array wie Blasen in einem Wasserglas aufsteigen, nennt man dieses Sortierverfahren *Bubblesort*.

BUBBLESORT

BUBBLESORT$(A = (A[1], \ldots, A[n]))$

```
1 for j ← 1 to n − 1 do
2     for i ← 1 to n − 1 do
3         if A[i] > A[i + 1] then Vertausche A[i] und A[i + 1].
```

ANALYSE

Wie viele Operationen führt der Algorithmus aus? Offenbar werden in jedem Fall $(n-1)^2 \in \Theta(n^2)$ viele Schlüssel miteinander verglichen. Im schlechtesten Fall (wenn A absteigend sortiert ist) wird zudem bei jedem Vergleich zweier Schlüssel eine Vertauschung vorgenommen, also gibt es im schlechtesten Fall $\Theta(n^2)$ viele Schlüsselvertauschungen. Im besten Fall (wenn A bereits sortiert ist) dagegen werden keine Schlüssel vertauscht.

VERBESSERUNG

Man könnte nun einwenden, dass die obige Implementierung von Bubblesort sehr naiv ist. So z.B. muss die innere Schleife eigentlich nur bis $n-j$ laufen, denn die Schlüssel an den Positionen $n-j+2, \ldots, n$ befinden sich bereits an der korrekten Position. Wir können uns ausserdem merken, ob bei einem Durchlauf der inneren Schleife überhaupt eine Vertauschung vorgenommen wurde. Ist dies nämlich nicht

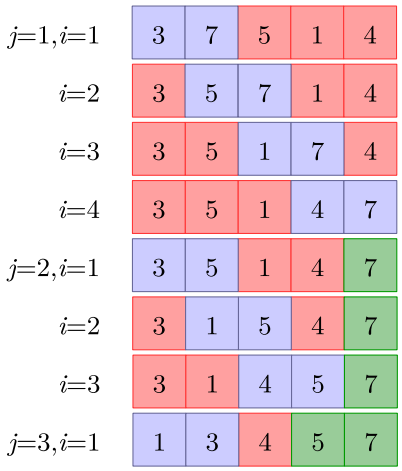


Abb. 2.3 Ausführung in Bubblesort auf dem Array $(3, 7, 5, 1, 4)$. Blau markierte Schlüssel werden in diesem Schritt miteinander verglichen und ggf. vertauscht (dargestellt ist die Situation nach der möglichen Vertauschung). Grün markierte Schlüssel sind bereits an der finalen Position. Der Einfachheit halber sind die letzten Schritte, in denen sich das Array nicht mehr ändern, nicht dargestellt.

der Fall, dann ist das Array bereits fertig sortiert und der Algorithmus kann vorzeitig beendet werden. Obwohl damit die Anzahl der Schlüsselvergleiche im besten Fall von $\Theta(n^2)$ auf $n-1$ sinkt, ändert dies nichts an den $\Theta(n^2)$ Schlüsselvergleichen bzw. -vertauschungen im schlechtesten Fall.

2.3.2 Sortieren durch Auswahl

Betrachten wir noch einmal Bubblesort. In der j-ten Phase wird jeweils das j-grösste Element an die korrekte Position $A[n-j+1]$ gebracht. Das heisst, nach Abschluss von j Phasen befinden sich an den letzten j Positionen des Arrays bereits die j grössten Schlüssel.

Das Problem bei Bubblesort war, dass u.U. sehr viele Schlüsselvertauschungen vorgenommen werden, nämlich $\Theta(n^2)$ im schlechtesten Fall. Der Grund ist, dass wir einen Schlüssel immer nur lokal mit seinem Nachbarn austauschen. Stattdessen könnten wir auch induktiv wie folgt vorgehen. Angenommen, die letzten j Positionen des Arrays enthalten bereits die j grössten Schlüssel in der richtigen Reihenfolge, d.h., sie befinden sich schon an ihrem endgültigen Platz. Wir suchen jetzt den $(j+1)$-grössten Schlüssel. Dieser ist der Schlüssel mit dem grössten Wert im Bereich $A[1], \ldots, A[n-j]$. Dann tauschen wir ihn und den Schlüssel auf der Position $A[n-j]$ aus, erhöhen j um 1 und fahren so fort, bis das Array sortiert ist. Dieses Sortierverfahren ist unter dem Namen *Sortieren durch Auswahl* bekannt.

Induktives Vorgehen

SelectionSort(A) — Sortieren durch Auswahl

```
1 for k ← n downto 2 do          ▷ k Schlüssel sind noch nicht sortiert
2     Berechne den Index i des maximalen Schlüssels in A[1], ..., A[k].
3     Vertausche A[i] und A[k].   ▷ A[k] enthält (n − k + 1)-grössten Schl.
```

Haben wir gegenüber Bubblesort etwas gewonnen? Zunächst einmal beobachten wir, dass zur Berechnung des maximalen Schlüssels im Bereich $A[1], \ldots, A[k]$ immer $k-1$

Analyse

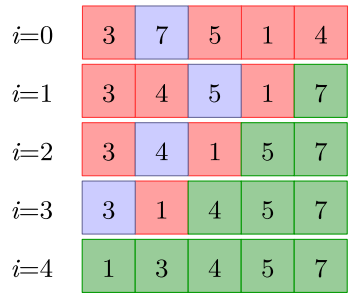


Abb. 2.4 Ausführung von Sortieren durch Auswahl auf dem Array $(3, 7, 5, 1, 4)$. Blau markierte Schlüssel sind das Maximum unter den noch zu sortierenden Schlüsseln. Grün markierte Schlüssel sind bereits an der finalen Position.

Vergleiche anfallen. Damit werden also grundsätzlich $\sum_{k=1}^{n}(k-1) \in \Theta(n^2)$ viele Schlüssel miteinander verglichen, was gegenüber Bubblesort keine Verbesserung darstellt. Allerdings wird in jedem Durchlauf der Schleife nur eine Schlüsselvertauschung vorgenommen, damit beträgt die Anzahl der Schlüsselvertauschungen lediglich $n-1$.

VERBESSERUNG Führen wir zudem die Vertauschung nur dann aus, wenn $i \neq k$ gilt, dann werden im besten Fall überhaupt keine Schlüssel mehr vertauscht, und im schlechtesten Fall weiterhin nur $n-1$.

VARIANTE Sortieren durch Auswahl wurde hier so beschrieben, dass der sortierte Bereich am Ende wächst, d.h., nach k Durchläufen der äusseren Schleife die letzten k Positionen im Array die jeweils k grössten Schlüssel speichern. Natürlich kann man auch in den ersten k Positionen die k kleinsten Schlüssel speichern, und im Bereich $A[k+1], \ldots, A[n]$ das Minimum suchen und an die Stelle $A[k+1]$ tauschen. Dies ändert weder die Korrektheit, noch die Laufzeit des Verfahrens.

2.3.3 Sortieren durch Einfügen

Eine andere Art, einen induktiven Sortieralgorithmus zu entwerfen, besteht in der Annahme, dass das Array bis zu einer gewissen Position k bereits sortiert ist, diese Positionen aber nicht unbedingt schon die k kleinsten Schlüssel enthalten. Anders als bei (der zuletzt diskutierten Variante von) Sortieren durch Auswahl befinden sich dort also irgendwelche Schlüssel des Arrays, die aber in aufsteigender Reihenfolge angeordnet sind. Der nächste Schlüssel $A[k+1]$ muss demnach an der richtigen Position in $A[1], \ldots, A[k]$ eingefügt werden (oder an der Position $k+1$ verbleiben, falls $A[k+1] \geq A[k]$ ist). Um die korrekte Position des Schlüssels $A[k+1]$ im bereits sortierten Bereich $A[1], \ldots, A[k]$ zu finden, führen wir eine binäre Suche nach $A[k+1]$ auf $A[1], \ldots, A[k]$ durch. Diese gibt den Index $i \in \{1, \ldots, k+1\}$ der Position, an die $A[k+1]$ eingefügt werden muss, zurück. Es gilt dann offenbar $A[i-1] \leq A[k+1] \leq A[i]$ (bzw. $A[k+1] \leq A[i]$ falls $i = 1$ ist). Da ein Array ein starre Struktur hat, können wir nicht einfach $A[k+1]$ zwischen die bestehenden Schlüssel $A[i]$ und $A[i+1]$ einfügen. Stattdessen muss $A[k+1]$ zunächst in einer temporären Variable b zwischengespeichert werden, $A[k]$ auf $A[k+1]$ kopiert werden, $A[k-1]$ auf $A[k]$, usw., und abschliessend kann $A[i]$ auf b gesetzt und b verworfen werden.

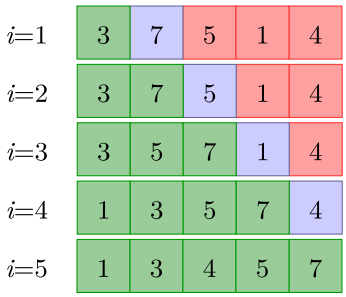


Abb. 2.5 Ausführung von Sortieren durch Einfügen auf dem Array $(3, 7, 5, 1, 4)$. Blau markierte Schlüssel sind im nächsten Schritt zu sortieren. Die grün markierten Schlüssel zeigen den bereits sortierten Bereich an.

Sortieren durch Einfügen

```
InsertionSort(A = (A[1], ..., A[n]))
1 for k ← 1 to n − 1 do                ▷ k Schlüssel sind bereits sortiert
2     Benutze binäre Suche auf A[1], ..., A[k], um die Position i
      zu finden, an die A[k + 1] eingefügt werden muss.
3     b ← A[k + 1]                     ▷ Speichere A[k + 1] in b zwischen
4     for j ← k downto i do            ▷ Verschiebe A[i], ..., A[k]
5         A[j + 1] ← A[j]              ▷ auf A[i + 1], ..., A[k + 1]
6     A[i] ← b                         ▷ Speichere A[k + 1] an A[i]
```

Analyse

Um die Anzahl der Schlüsselvergleiche zu analysieren, erinnern wir uns zunächst, dass binäre Suche auf k Schlüsseln höchstens Zeit $\mathcal{O}(\log k)$ benötigt. Es gibt also eine Konstante a, sodass die binäre Suche höchstens $a \log(k)$ viele Vergleiche ausführt. Damit führt Sortieren durch Einfügen grundsätzlich insgesamt höchstens

$$\sum_{k=1}^{n-1} a \log(k) = a \log((n-1)!) \in \mathcal{O}(n \log n) \tag{37}$$

viele Vergleiche aus (die Abschätzung der Summe findet sich in der Mitschrift zu den mathematischen Grundlagen). Im besten Fall werden nur $\Theta(n)$ viele Schlüssel bewegt (und sogar keiner, wenn man nur dann verschiebt, wenn $i \neq k+1$ ist). Im schlechtesten Fall allerdings wird in jedem Schleifendurchlauf der Schlüssel $A[k+1]$ an die erste Stelle verschoben, was insgesamt $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$ viele Verschiebungen zur Folge hat. Wir sehen aber auch, dass die Laufzeit wesentlich besser ist, wenn das Array bereits vollständig bzw. zum grossen Teil vorsortiert ist.

2.4 Heapsort

Wir erinnern uns kurz an *Sortieren durch Auswahl*. Die quadratische Laufzeit dieses Verfahrens ergab sich aus der linearen Laufzeit zur Bestimmung des maximalen Schlüssels. Wenn wir also das Maximum effizienter bestimmen könnten, dann ergäbe sich ein Sortierverfahren, das mit weniger als quadratischer Zeit auskommt. Der im Folgenden vorgestellte *Heap* erlaubt die Extraktion des maximalen Schlüssels in Zeit $\mathcal{O}(\log n)$, wenn n Schlüssel verwaltet werden.

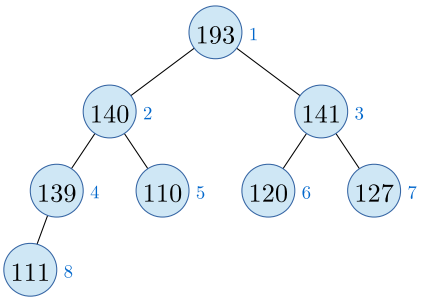


Abb. 2.6 Darstellung des Max-Heaps $A = (193, 140, 141, 139, 110, 120, 127, 111)$ als binärer Baum. Der Index rechts neben einem Knoten mit Schlüssel k entspricht der Position von k im Array A.

MAX-HEAP

Heaps Ein Array $A = (A[1], \ldots, A[n])$ mit n Schlüsseln heisst *Max-Heap*, wenn alle Positionen $k \in \{1, \ldots, n\}$ die *Heap-Eigenschaft*

HEAP-EIGENSCHAFT

$$\big((2k \leq n) \Rightarrow (A[k] \geq A[2k])\big) \text{ und } \big((2k+1 \leq n) \Rightarrow (A[k] \geq A[2k+1])\big) \qquad (38)$$

erfüllen. Obwohl ein Max-Heap intern als Array gespeichert wird, kann er als *binärer Baum* visualisiert werden, wobei jeder Schlüssel in genau einem *Knoten* gespeichert wird (vgl. Abbildung 2.6). Wir beobachten, dass die Nachfolger(knoten) des in A an Position k gespeicherten Knotens im Array an den Positionen $2k$ sowie $2k+1$ gespeichert sind, sofern diese existieren. Die Heap-Eigenschaft besagt damit anschaulich, dass für einen Knoten mit Schlüssel x die Schlüssel der entsprechenden Nachfolger höchstens so gross wie x selbst sind. Folglich ist der maximale Schlüssel des Heaps in $A[1]$ bzw. im obersten Knoten gespeichert. Weiterhin sehen wir, dass jeder unter einem Knoten gespeicherte Teilbaum ebenfalls als Heap interpretiert werden kann.

AUSLESEN DES MAXIMUMS

EXTRAKTION DES MAXIMUMS

Während das reine Auslesen des maximalen Schlüssels also einfach ist, ist nicht offensichtlich, wie er effizient aus dem Heap extrahiert (d.h. entfernt) werden kann. Das Problem liegt in der starren, durch das Array implizit vorgegebenen Struktur des Heaps. Wir lösen das Problem wie folgt. Sei m eine Variable, die die Anzahl der Schlüssel eines Heaps $(A[1], \ldots, A[m])$ speichert. Wir speichern zu extrahierenden Schlüssel $A[1]$, kopieren den in $A[m]$ gespeicherten Schlüssel nach $A[1]$ und verringern m um 1. Nun aber ist $(A[1], \ldots, A[m])$ im Allgemeinen kein Heap mehr, denn an der Wurzel ist die Heap-Eigenschaft verletzt (der dort gespeicherte Schlüssel ist kleiner als die Schlüssel der Nachfolger).

Daher wird der in der Wurzel gespeicherte Schlüssel mit dem *grösseren* der an den beiden Nachfolgern gespeicherten Schlüssel vertauscht. Somit ist die Heap-Eigenschaft an der Wurzel erfüllt, kann aber am entsprechenden Nachfolger verletzt sein. Folglich wird genau dort die Heap-Eigenschaft auf die gleiche Art wiederhergestellt, und die Wiederherstellung wird für die jeweiligen Nachfolger fortgesetzt, bis $(A[1], \ldots, A[m])$ wieder ein Heap ist. Man beachte, dass die Heap-Eigenschaft jeweils nur für den Nachfolger w_1 wiederhergestellt werden muss, dessen Schlüssel mit dem Schlüssel des jeweiligen Vorgängers v vertauscht wurde. Für den anderen Nachfolger w_2 (sofern existent) ist die Heap-Eigenschaft sowieso erfüllt, weil in dem unter w_2 gespeicherten Teilbaum überhaupt keine Schlüssel verändert wurden, und weil der in w_1 vorher gespeicherte Schlüssel grösser als der in w_2 gespeicherte Schlüssel ist.

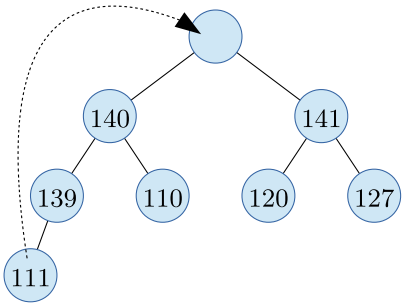


Abb. 2.7 Extraktion des Maximums (Schlüssel 193) aus dem in Abbildung 2.6 dargestellten Heap. Im ersten Schritt wird der in $A[8]$ gespeicherte Schlüssel nach $A[1]$ kopiert.

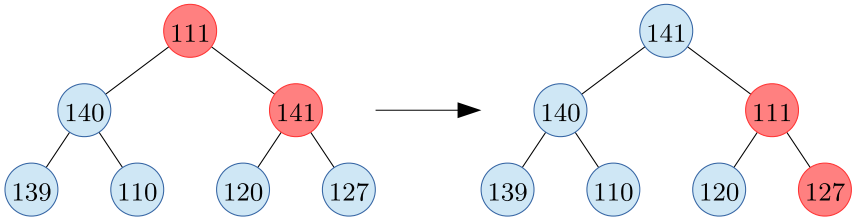


Abb. 2.8 Wiederherstellung der Heap-Eigenschaft für die in Abbildung 2.7 dargestellte Struktur. Zunächst werden die Schlüssel 111 und 141 vertauscht, danach 111 und 127.

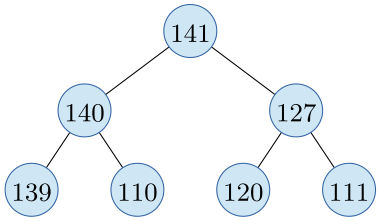


Abb. 2.9 Der rekonstruierte Heap $A = (141, 140, 127, 139, 110, 120, 111)$.

RESTORE-HEAP-CONDITION(A, i, m) — WIEDERHERSTELLUNG DES HEAPS

```
1 while 2 · i ≤ m do                          ▷ A[i] hat linken Nachfolger
2     j ← 2 · i                               ▷ A[j] ist linker Nachfolger
3     if j + 1 ≤ m then                       ▷ A[i] hat rechten Nachfolger
4         if A[j] < A[j + 1] then j ← j + 1   ▷ A[j] ist grösserer Nachfolger
5     if A[i] ≥ A[j] then STOP                ▷ Heap-Bedingung erfüllt
6     Vertausche A[i] und A[j]                ▷ Reparatur
7     i ← j                                   ▷ Weiter mit Nachfolger
```

LAUFZEIT

Die Laufzeit von RESTORE-HEAP-CONDITION ist offenbar in $\mathcal{O}(\log m)$: Jeder der Schritte 2–7 benötigt nur konstante Zeit, initial ist $i \geq 1$, Schritt 7 verdoppelt den Wert von i mindestens, und die Schleife stoppt sobald erstmalig $i > m/2$ ist. Die Anzahl der Schleifendurchläufe (und damit die Gesamtlaufzeit) ist also durch $\mathcal{O}(\log m)$ nach oben beschränkt. Schaut man etwas genauer hin, kann man sogar sehen, dass die Laufzeit sogar in $\mathcal{O}(\log(m/i))$ ist.

Erzeugung eines Heaps Im vorigen Abschnitt wurde gezeigt, wie der maximale Schlüssel effizient aus dem Heap extrahiert werden kann. Wie aber kann aus einer

Menge von n Schlüsseln $(A[1], \ldots, A[n])$ effizient ein Heap zu dieser Schlüsselmenge erstellt werden? Wir beobachten zunächst, dass für die Positionen $\lfloor n/2 \rfloor + 1, \ldots, n$ die Heap-Eigenschaft sofort gilt (anschaulich gesprochen besitzen diese keine Nachfolgerknoten und können als Heaps mit genau einem Schlüssel interpretiert werden). Für die Position $\lfloor n/2 \rfloor$ muss aber die Heap-Eigenschaft nicht notwendigerweise erfüllt sein. Wir rufen nun für $i \in \{\lfloor n/2 \rfloor, \ldots, 1\}$ sukzessive den Algorithmus RESTORE-HEAP-CONDITION(A, i, n) auf. Vor dem Aufruf mit Parameter i galt die Heap-Eigenschaft bereits für alle $k \in \{i+1, \ldots, n\}$, und nach Terminierung des Aufrufs gilt die Heap-Eigenschaft für alle $k \in \{i, \ldots, n\}$. Folglich ist nach dem letzten Aufruf mit dem Parameter $i = 1$ das Array $(A[1], \ldots, A[n])$ ein Max-Heap.

LAUFZEIT

Da wir zum Aufbauen des Heaps nur $\mathcal{O}(n)$ Mal RESTORE-HEAP-CONDITION aufrufen und jeder solche Aufruf nur Zeit $\mathcal{O}(\log n)$ benötigt, kann ein unsortiertes Array in Zeit $\mathcal{O}(n \log n)$ in einen Max-Heap überführt werden. Diese Schranke kann noch verbessert werden, denn wir hatten bereits im Vorfeld argumentiert, dass RESTORE-HEAP-CONDITION(A, i, n) nur Zeit $\mathcal{O}(\log(n/i))$ benötigt. Damit beträgt die Zeit zum Aufbauen des Heaps lediglich

$$C \sum_{i=1}^{\lfloor n/2 \rfloor} \ln(n/i) \leq C \sum_{i=1}^{\lceil n/2 \rceil} \ln(n/i) = C \ln \left(\prod_{i=1}^{\lceil n/2 \rceil} \frac{n}{i} \right) = C \ln \left(\frac{n^{\lceil n/2 \rceil}}{\prod_{i=1}^{\lceil n/2 \rceil} i} \right)$$

$$= C \ln \left(\frac{n^{\lceil n/2 \rceil}}{(\lceil n/2 \rceil)!} \right) = C \Big(\ln \left(n^{\lceil n/2 \rceil} \right) - \ln \left((\lceil n/2 \rceil)! \right) \Big) \tag{39}$$

$$\overset{(*)}{\leq} C \Big(\lceil n/2 \rceil \ln n - \lceil n/2 \rceil \ln(\lceil n/2 \rceil) + \lceil n/2 \rceil \Big) \tag{40}$$

$$\leq C \Big(\lceil n/2 \rceil \big(\ln n - \ln(n/2) \big) + \lceil n/2 \rceil \Big) \tag{41}$$

$$\leq C \Big(\lceil n/2 \rceil \big(\ln n - \ln n + \ln 2 \big) + \lceil n/2 \rceil \Big) \leq Cn \tag{42}$$

für eine geeignet gewählte Konstante C. Die Ungleichung $(*)$ gilt aufgrund der früher bewiesenen Abschätzung $\ln(n!) \geq n \ln(n) - n$.

Heapsort Sei $A = (A[1], \ldots, A[n])$ das zu sortierende Array. Zur Sortierung von A benutzen wir nun zunächst das im vorigen Abschnitt vorgestellte Verfahren zur Erzeugung eines Heaps aus A, und danach extrahieren wir $n-1$ Mal das Maximum aus dem Heap und tauschen es an den Anfang des bereits sortierten Bereichs (am Ende des Arrays).

HEAPSORT

HEAPSORT(A)

```
1 for i ← ⌊n/2⌋ downto 1 do
2     RESTORE-HEAP-CONDITION(A, i, n)
3 for m ← n downto 2 do
4     Vertausche A[1] und A[m]
5     RESTORE-HEAP-CONDITION(A, 1, m − 1)
```

Die Korrektheit des Vorgehens haben wir bereits vorher erläutert. Wir sehen auch, dass die Anzahl der verglichenen und vertauschten Schlüssel stets in $\mathcal{O}(n \log n)$ liegt, denn es gibt $\mathcal{O}(n)$ Aufrufe von RESTORE-HEAP-CONDITION, von denen jeder $\mathcal{O}(\log n)$ viele Schlüssel vergleicht bzw. vertauscht. Wir haben also endlich ein Sortierverfahren entworfen, das selbst im schlechtesten Fall Laufzeit $\mathcal{O}(n \log n)$ hat.

2.5 Mergesort

Idee Schon beim *Maximum-Subarray-Problem* haben wir gesehen, dass Divide-and-Conquer helfen kann, effiziente Algorithmen zu entwerfen. Dieses Prinzip des rekursiven Aufteilens führt zu folgender Idee für ein Sortierverfahren: Die Folge $A[1], \ldots, A[n]$ wird in die zwei annähernd gleich grossen Teilfolgen $A[1], \ldots, A[\lfloor n/2 \rfloor]$ sowie $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ aufgeteilt, die rekursiv sortiert werden. Nach der Sortierung der Teilfolgen werden diese wieder zu einer *sortierten* Gesamtfolge der Länge n zusammengefügt. Wir zeigen zunächst, wie diese *Verschmelzung* sortierter Teilfolgen realisiert werden kann. Danach beschreiben wir Sortierverfahren, die bereits sortierte Teilfolgen verschmelzen und so eine insgesamt sortierte Folge erzeugen.

Verschmelzen sortierter Teilfolgen Seien F_1 und F_2 zwei sortierte Teilfolgen. Die sortierte Gesamtfolge F wird erzeugt, indem F_1 und F_2 mit zwei Zeigern von vorn nach hinten durchlaufen werden. Dies geschieht mithilfe von zwei Variablen i und j, die wir mit 1 initialisieren. Ist $F_1[i] < F_2[j]$, dann wird $F_1[i]$ das nächste Element von F, und der Zeiger i wird um 1 erhöht. Ansonsten wird $F_2[j]$ das nächste Element von F, und j wird um 1 erhöht. Dieses Vorgehen wird wiederholt, bis entweder F_1 oder F_2 vollständig durchlaufen sind. Danach wird die noch nicht erschöpfte Folge komplett an das Ende von F gehängt.

Der im Folgenden vorgestellte Algorithmus MERGE realisiert die Verschmelzung zweier bereits sortierter Teilfolgen. Diesem Algorithmus werden die vier Parameter A, left, middle und right übergeben. Dabei findet sich die bereits sortierte Teilfolge F_1 in $A[\text{left}], \ldots, A[\text{middle}]$ und F_2 in $A[\text{middle}+1], \ldots, A[\text{right}]$. Die verschmolzene Folge F soll schliesslich in $A[\text{left}], \ldots, A[\text{right}]$ gespeichert werden. Da dies genau die Positionen in A sind, an denen F_1 und F_2 gespeichert sind, muss die sortierte Folge zunächst in einem Hilfsarray B der Länge right−left+1 gespeichert werden. Die Inhalte von B werden dann im letzten Schritt auf die entsprechenden Positionen im Array A kopiert.

VERSCHMELZEN DER TEILFOLGEN

MERGE(A, left, middle, right)

```
1 B ← new Array[right-left+1]          ▷ Hilfsarray zum Verschmelzen
2 i ← left; j ← middle+1; k ← 1        ▷ Zeiger zum Durchlaufen
3 ▷ Wiederhole, solange beide Teilfolgen noch nicht erschöpft sind
  while (i ≤ middle) and (j ≤ right) do
4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1
5     else B[k] ← A[j]; j ← j + 1
6     k ← k + 1
7 ▷ Falls die erste Teilfolge noch nicht erschöpft ist, hänge sie hinten an
  while i ≤ middle do B[k] ← A[i]; i ← i + 1; k ← k + 1
8 ▷ Falls die zweite Teilfolge noch nicht erschöpft ist, hänge sie hinten an
  while j ≤ right do B[k] ← A[j]; j ← j + 1; k ← k + 1
9 ▷ Kopiere Inhalte von B nach A zurück
  for k ← left to right do A[k] ← B[k − left + 1]
```

KORREKTHEIT **Lemma 2.1.** *Seien* $(A[1], \ldots, A[n])$ *ein Array,* $1 \leq l \leq m \leq r \leq n$ *und die Teilarrays* $(A[l], \ldots, A[m])$ *als auch* $(A[m+1], \ldots, A[r])$ *bereits sortiert. Nach dem Aufruf von* MERGE(A, l, m, r) *ist* $(A[l], \ldots, A[r])$ *komplett sortiert.*

Beweis. Wir zeigen zunächst induktiv die folgende Aussage: *Nach* k *Durchläufen der Schleife in Schritt 3 ist* $(B[1], \ldots, B[k])$ *sortiert, und es sind* $B[k] \leq B[i]$*, falls* $i \leq m$*, sowie* $B[k] \leq B[j]$*, falls* $j \leq r$*.*

Induktionsanfang ($k = 0$): Vor dem ersten Schleifendurchlauf hat das Array $(B[1], \ldots, B[0])$ Länge 0 und die Aussage gilt trivialerweise. (!)

Induktionshypothese: Sei nach k Durchläufen der Schleife $(B[1], \ldots, B[k])$ sortiert, und $B[k] \leq A[i]$, falls $i \leq m$, sowie $B[k] \leq A[j]$, falls $j \leq r$.

Induktionsschluss ($k \to k+1$): Betrachte den $(k+1)$-ten Durchlauf der Schleife in Schritt 3. Es seien o.B.d.A. $A[i] \leq A[j]$, $i \leq m$ und $j \leq r$. Nach der Induktionshypothese ist $(B[1], \ldots, B[k])$ sortiert, und $B[k] \leq A[i]$. Wird also $B[k+1] \leftarrow A[i]$ gesetzt, dann ist $(B[1], \ldots, B[k+1])$ noch immer sortiert, und $B[k+1] = A[i] \leq A[i+1]$, falls $(i+1) \leq m$, sowie $B[k+1] \leq A[j]$, falls $j \leq r$. Anschliessend werden sowohl i als auch k um 1 erhöht, also gilt die Aussage auch nach dem $(k+1)$-ten Schleifendurchlauf. Der Fall $A[j] < A[i]$ wird analog bewiesen.

Nach der Terminierung der Schleife in Schritt 3 gilt entweder $i > m$ oder $j > r$. Also wird *genau eine* der Schleifen in den Schritten 7 und 8 ausgeführt. Sei o.B.d.A. $i \leq m$. Dann ist nach k Durchläufen der Schleife im dritten Schritt $B[k] \leq A[i]$, und da $(A[i], \ldots, A[m])$ sortiert ist, können die verbleibenden Elemente von $(A[i], \ldots, A[m])$ an das Ende von B gehängt werden, und nach Schritt 8 ist $(B[1], \ldots, B[r-l+1])$ komplett sortiert. ∎

LAUFZEIT **Lemma 2.2.** *Seien* $(A[1], \ldots, A[n])$ *ein Array,* $1 \leq l < r \leq n$*,* $m = \lfloor (l+r)/2 \rfloor$ *und die Teilarrays* $(A[l], \ldots, A[m])$ *als auch* $(A[m+1], \ldots, A[r])$ *bereits sortiert. Im Aufruf von* MERGE(A, l, m, r) *werden* $\Theta(r-l)$ *viele Schlüsselbewegungen und Vergleiche ausgeführt.*

Beweis. Der Aufwand zur Initialisierung des Arrays der Grösse $(r-l+1)$ im ersten Schritt beträgt $\Theta(r-l)$. Die Schleifen in den Schritten 3 und 9 werden $\Theta(r-l)$ Mal durchlaufen, die Schleifen in den Schritten 7 und 8 nur höchstens $\mathcal{O}(r-l)$ Mal. Da für alle anderen Schritte nur Zeit $\mathcal{O}(1)$ anfällt, ergibt sich eine Gesamtlaufzeit von $\Theta(r-l)$. In jedem Durchlauf der Schleife in Schritt 3 wird genau ein Schlüsselvergleich ausgeführt. Ausserdem wird jedes Element genau einmal nach B und genau einmal zurück nach A kopiert. Damit liegt die Anzahl der bewegten und verglichenen Schlüssel ebenfalls in $\Theta(r-l)$. ∎

2.5.1 Rekursives 2-Wege-Mergesort

Der im letzten Abschnitt vorgestellte Algorithmus zur Verschmelzung zweier sortierter Teilfolgen kann nun benutzt werden, um ein Array rekursiv zu sortieren. Dem im Folgenden vorgestellten Algorithmus MERGESORT werden die drei Parameter A, left und right übergeben, und die Aufgabe besteht in der Sortierung des Teilarrays

$(A[\text{left}], \ldots, A[\text{right}])$. Zur Sortierung des gesamten Arrays $A = (A[1], \ldots, A[n])$ wird dann MERGESORT$(A, 1, n)$ aufgerufen.

REKURSIVES MERGESORT

MERGESORT$(A, \text{left}, \text{right})$

```
1 if left < right then
2     middle ← ⌊(left + right)/2⌋          ▷ Mittlere Position
3     MERGESORT(A, left, middle)           ▷ Sortiere vordere Hälfte von A
4     MERGESORT(A, middle + 1, right)      ▷ Sortiere hintere Hälfte von A
5     MERGE(A, left, middle, right)        ▷ Verschmilz Teilfolgen
```

LAUFZEIT

Zur Bestimmung der Laufzeit von Mergesort stellen wir eine Rekursionsgleichung $C(n)$ für den Aufwand zur Sortierung eines Arrays mit n Schlüsseln auf. Für $n = 1$ ist offenbar $C(1) \in \Theta(1)$. Zum Verschmelzen zweier annähernd gleich langer Teilfolgen mit Gesamtlänge $n/2$ werden nach Lemma 2.2 $\Theta(n)$ Schlüsselvergleiche benötigt. Daher führt Mergesort auf einer Eingabe der Länge n stets

$$C(n) = C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n) \tag{43}$$

viele Schlüsselvergleiche und -bewegungen aus.

2.5.2 Reines 2-Wege-Mergesort

Die rekursiven Aufrufe im Mergesort machen gar nicht viel ausser mit den Indizes herumzurechnen. Stattdessen kann man Mergesort auch Bottom-up ausführen, so wie mit der Struktur des Heaps. Dazu sortieren wir zuerst die Teilstücke der Länge 2, dann kombinieren (d.h., verschmelzen) wir immer je zwei Stücke zu Teilstücken der Länge 4, usw. So gehen wir durch alle Zweierpotenzen, bis wir die Länge des Arrays erreichen.

REINES 2-WEGE-MERGESORT

STRAIGHTMERGESORT(A)

```
1 length ← 1                              ▷ Länge bereits sortierter Teilfolgen
2 while length < n do                     ▷ Verschmilz Folgen d. Länge length
3     right ← 0                           ▷ A[1], ..., A[right] ist abgearbeitet
4     while right+length < n do           ▷ Es gibt noch mind. zwei Teilfolgen
5         left ← right+1                  ▷ Linker Rand der ersten Teilfolge
6         middle ← left+length−1          ▷ Rechter Rand der ersten Teilfolge
7         right ← min(middle+length, n)   ▷ Rechter Rand der zweite Teilfolge
8         MERGE(A, left, middle, right)   ▷ Verschmilz beide Teilfolgen
9     length ← length·2                   ▷ Verdoppelte Länge der Teilfolgen
```

LAUFZEIT

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus.

2.5.3 Natürliches 2-Wege-Mergesort

Natürliches 2-Wege-Mergesort vermeidet wie reines 2-Wege-Mergesort rekursive Aufrufe, beginnt aber initial mit der Verschmelzung von möglichst langen bereits sortierten Teilfolgen, während reines 2-Wege-Mergesort immer mit sortierten Folgen der Länge 1 beginnt. Das zu sortierende Array A bestehe nun aus k längsten bereits sortierten Teilfolgen (sog. *Runs*) $R_1, \ldots, R_k$. Ein neuer Run beginnt also genau dann, sobald direkt hinter einem Schlüssel ein kleinerer steht. Natürliches 2-Wege-Mergesort verschmilzt R_1 mit R_2, danach R_3 mit R_4 usw. Dieses Verfahren wird so lange wiederholt, bis es nur noch einen Run (das sortierte Array A) gibt.

Natürliches 2-Wege-Mergesort

NaturalMergesort(A)

```
 1 repeat
 2     right ← 0                          ▷ Elemente bis A[right] sind bearbeitet
 3     while right < n do                 ▷ Finde und verschmilz die nächsten Runs
 4         left ← right+1                 ▷ Linker Rand des ersten Runs
 5         middle ← left                  ▷ A[left], ..., A[middle] ist bereits sortiert
 6         while (middle < n) and A[middle + 1] ≥ A[middle] do
 7             middle ← middle+1          ▷ Ersten Run um ein Element vergrössern
 8         if middle < n then             ▷ Es gibt einen zweiten Run
 9             right ← middle+1           ▷ Rechter Rand des zweiten Runs
10             while (right < n) and A[right + 1] ≥ A[right] do
11                 right ← right+1        ▷ Zweiten Run um ein Element vergrössern
12             Merge(A, left, middle, right)
13         else right ← n                 ▷ Es gibt keinen zweiten Run
14 until left = 1
```

Laufzeit

Genau wie rekursives und reines 2-Wege-Mergesort führt natürliches Mergesort im schlechtesten und im durchschnittlichen Fall $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus. Im besten Fall dagegen ist das Array bereits sortiert, d.h. es besteht nur aus einem Run. Dann ist $A[\text{middle} + 1] \geq A[\text{middle}]$ in Schritt 6 immer wahr, und die Schleife terminiert für middle $= n$. Folglich wird in Schritt 13 right $\leftarrow n$ gesetzt, und die Schleife in Schritt 3 terminiert nach einem Durchlauf. Es werden also keine keine Schlüssel bewegt, und die Anzahl der Vergleiche beträgt $n - 1$.

2.6 Quicksort

Rückblick

Rückblick Erinnern wir uns kurz zurück an Mergesort. Dort wird das Array in der Mitte geteilt, sowohl der linke als auch der rechte Teil jeweils rekursiv sortiert und dann die sortierten Teilfolgen zu einer sortierten Gesamtfolge verschmolzen. Wir hatten bereits gesehen, dass sowohl die Anzahl der Schlüsselvergleiche als auch die Anzahl der Bewegungen im schlimmsten Fall durch $\mathcal{O}(n \log n)$ nach oben beschränkt ist, dafür aber das Verschmelzen $\Theta(n)$ viel Zusatzspeicher benötigt. Um mit weniger Zusatzspeicher auszukommen, werden wir nun überlegen, wie wir das Verschmelzen umgehen können ohne dabei die Korrektheit des Verfahrens aufzuheben.

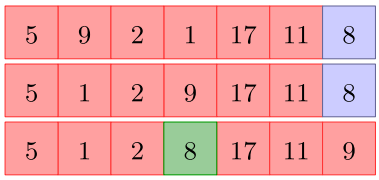


Abb. 2.10 Umordnung des Arrays $A = (5, 9, 2, 1, 17, 11, 8)$ mit PARTITION. Als Pivotelement wird 8 gewählt (blau dargestellt). Die Schlüssel 9 und 1 befinden sich in falscher Reihenfolge und werden daher vertauscht. Am Ende des Aufteilungsschritts werden die Schlüssel 9 und 8 vertauscht und die Position 4 zurückgeliefert (denn dort befindet sich das Pivotelement). Der Schlüssel 8 befindet sich dann an der korrekten Position im Array, und die Teilfolgen $(A[1], A[2], A[3])$ sowie $(A[5], A[6], A[7])$ werden rekursiv sortiert.

SORTIEREN OHNE ZU VERSCHMELZEN

Idee Das spätere Verschmelzen könnten wir uns sparen, wenn wir bereits wüssten, dass alle Schlüssel in der linken Teilfolge kleiner als alle Schlüssel in der rechten Teilfolge wären. Um dies sicherzustellen, investieren wir ein wenig mehr Arbeit in die Aufteilung des Arrays als vorher (in Mergesort ist die Aufteilung in zwei gleich grosse Teile trivial). Dazu nehmen wir für den Moment an, wir hätten bereits eine Methode PARTITION, die ein Array A im Bereich $l, \ldots, r$ so umsortiert, dass sich an *irgendeiner* Position $k \in \{l, \ldots, r\}$ bereits der korrekte Schlüssel befindet, alle Schlüssel links der Position k kleiner als $A[k]$ und alle Schlüssel rechts der Position k grösser als $A[k]$ sind (siehe Abbildung 2.10). Wir nehmen ausserdem an, PARTITION würde die Position k zurückliefern, die das Array in zwei Teile teilt. Dann könnten wir wie folgt rekursiv sortieren, ohne die sortierten Teilfolgen verschmelzen zu müssen:

HAUPTALGORITHMUS

QUICKSORT(A, l, r)

```
1 if l < r then                    ▷ Array enthält mehr als einen Schlüssel
2     k = PARTITION(A, l, r)       ▷ Führe Aufteilung durch
3     QUICKSORT(A, l, k − 1)       ▷ Sortiere linke Teilfolge rekursiv
4     QUICKSORT(A, k + 1, r)       ▷ Sortiere rechte Teilfolge rekursiv
```

IDEE ZUR AUFTEILUNG

PIVOTELEMENT

Aufteilungsschritt Wie aber kann das Array geeignet aufgeteilt werden? Es ist schwierig, das Array in zwei exakt gleich grosse Teile zu teilen, sodass alle Schlüssel links kleiner als alle Schlüssel rechts sind. Daher wählt man einfach *irgendeinen* Schlüssel p aus (das sog. *Pivotelement*), prüft dann für die anderen Schlüssel, ob sie kleiner oder grösser als p sind, und ordnet sie entsprechend um. Als Pivotelement p wählen wir den Schlüssel am rechten Rand des zu sortierenden Bereichs. Wir durchlaufen dann das Array zunächst von links, bis wir einen Schlüssel $A[i]$ finden, der grösser als p ist. Dann durchlaufen wir das Array von rechts, bis wir einen Schlüssel $A[j]$ finden, der kleiner als p ist. Ist nun $i < j$, dann sind sowohl $A[i]$ als auch $A[j]$ jeweils in der falschen Seite des Arrays und werden daher vertauscht. Dann fahren wie soeben beschrieben fort, bis $i \geq j$ ist. Nun muss nur noch das Pivotelement korrekt positioniert werden. Dazu vertauschen wir $A[i]$ mit p und sind fertig, denn alle Schlüssel links vom Pivotelement p (das sich jetzt an Position i befindet) sind kleiner als p, und alle Schlüssel rechts von p sind grösser als p. Der folgende Algorithmus realisiert die Aufteilung von $(A[l], \ldots, A[r])$.

Aufteilungsschritt

PARTITION(A, l, r)

```
1  i = l
2  j = r − 1
3  p = A[r]
4  repeat
5      while i < r and A[i] < p do i = i + 1
6      while j > l and A[j] > p do j = j − 1
7      if i < j then Vertausche A[i] und A[j]
8  until i ≥ j
9  Tausche A[i] und A[r]
10 return i
```

Korrektheit Zum Nachweis der Korrektheit beobachten wir zunächst, dass es ausreichend ist, sich auf den Fall $l < r$ zu beschränken (denn nur für solche Indizes wird PARTITION im Hauptalgorithmus QUICKSORT aufgerufen). Ausserdem nehmen wir wie immer an, dass alle Schlüssel im zu sortierenden Array A verschieden sind. Damit kann insbesondere das Pivotelement p nur einmal vorkommen.

Wir sehen schonmal, dass die Schritte 5–7 auf jeden Fall dann korrekt sind, wenn am Ende $i < j$ ist. Da j initial $r - 1$ ist und danach höchstens noch kleiner wird, gilt also $i < j < n$, also enthalten weder $A[i]$ noch $A[j]$ das Pivotelement p (denn dieses befindet sich ja auf Position r). Sind die Schleifen in den Schritten 5 und 6 abgearbeitet, dann ist $A[j] < p < A[i]$, und damit sind sowohl $A[j]$ als auch $A[i]$ auf der falschen Seite im Array und dürfen vertauscht werden (was Schritt 7 tut).

Gilt dagegen nach Ablauf der Schritte 5–7 $i \geq j$, dann zeigt $A[i]$ auf jeden Fall auf ein Element, welches nicht kleiner als p ist (unter der Annahme, dass alle Schlüssel verschieden sind, gilt $A[i] < p$ genau dann, wenn $i < r$ gilt). Zudem wissen wir, dass $A[i'] < p$ für alle $i' < i$ gilt, und $A[i'] > p$ für alle $i' > i$ mit $i' \neq r$ gilt. Tauschen wir also nach der Terminierung der äusseren Schleife in Schritt 8 die Schlüssel $A[i]$ und $A[r]$, dann enthält die Position i das Pivotelement p, alle Schlüssel links von i sind kleiner als p und alle Schlüssel rechts von i sind grösser als p, und der Aufteilungsschritt wurde korrekt durchgeführt.

Vergleiche in Schritt 5 **Praktische Überlegungen** In Schritt 5 des Aufteilungsschritts prüfen wir ja zunächst, ob $i < r$ gilt, und ist dies der Fall, dann prüfen wir weiterhin, ob zusätzlich $A[i] < p$ gilt. Erst dann wird i erhöht. Schaut man nun genauer hin, dann bemerkt man, dass es ausreichend ist, lediglich $A[i] < p$ zu prüfen, denn für $i = r$ ist $A[i] = p$, folglich ist der Vergleich $A[i] < p$ für $i = r$ *immer* falsch und die Schleife terminiert. Das Pivotelement dient also quasi als "Stopper".

Vergleiche in Schritt 6 Können wir einen ähnlichen Trick für die Schleife in Schritt 6 anwenden, um den Vergleich $j > l$ zu eliminieren? Wir könnten vor dem Aufruf von PARTITION(A, l, r) den Wert von $A[l-1]$ in einer temporären Variablen x speichern, dann $A[l-1] = -\infty$ setzen (wobei $-\infty$ für einen Schlüssel steht, der kleiner als alle im Array vorkommenden Schlüssel ist) und nach der Terminierung von PARTITION(A, l, r) wieder $A[l-1] = x$ setzen. Aufpassen müssen wir nur für $l = 1$, denn oftmals haben wir auf das 0-te Element keinen Zugriff (bzw. auf das (-1)-te Element, wenn das Array von 0 bis $n - 1$ indiziert ist).

Abb. 2.11 Wurde ein Array so umgeordnet, dass nach dem Aufteilungsschritt das obige Array vorliegt (wobei als Pivotelement 11 gewählt worden ist), dann wurden maximal zwei Schlüssel miteinander vertauscht.

Mehrfach vorkommende Schlüssel

Sortieren von wenigen Schlüsseln

Eine praktische Implementierung von Partition müsste zudem berücksichtigen, dass in der Realität Schlüssel durchaus mehrfach vorkommen können. Eine solche Anpassung ist zwar konzeptionell nicht schwierig, aber recht technisch, weshalb wir dies in diesem Skript nicht weiter verfolgen wollen. Zudem würde man Quicksort nicht rekursiv fortsetzen, wenn der zu sortierende Bereich nur wenige Schlüssel umfasst. Stattdessen sollte man besser ein elementares Sortierverfahren wie etwa Sortieren durch Einfügen anwenden.

Anzahl verglichener Schlüssel Der Aufteilungsschritt selbst vergleicht nur $\mathcal{O}(r-l)$ viele Schlüssel miteinander. Wie viele Schlüssel der gesamte Quicksort-Algorithmus insgesamt miteinander vergleicht, hängt offenbar stark von der Wahl des Pivotelements ab. Im besten Fall wird das Pivotelement immer so gewählt, dass wir zwei möglichst gleich grosse Teile entstehen, und die Anzahl insgesamt verglichener Schlüssel bei einer Eingabe der Länge n beträgt

Anzahl verglichener Schlüssel im besten Fall

$$T(n) = 2T(n/2) + c \cdot n, T(1) = 0 \tag{44}$$

für eine geeignete Konstante c, was $T(n) \in \mathcal{O}(n \log n)$ impliziert. Wählen wir aber in jedem Aufteilungsschritt ein ungünstiges Pivotelement (nämlich den kleinsten oder den grössten Schlüssel), dann erzeugen wir eine Teilfolge der Länge 0 und eine der Länge $n-1$. Folglich vergleicht Quicksort im schlimmsten Fall also

Anzahl verglichener Schlüssel im schlechtesten Fall

$$T'(n) = T'(n-1) + c \cdot n, T(1) = 0, \tag{45}$$

d.h. $\Theta(n^2)$ viele Schlüssel! Erschwerend kommt dazu, dass solch ein ungünstiger Fall gar nicht allzu schwierig zu konstruieren ist: Da im Aufteilungsschritt immer den am weitesten rechts stehenden Schlüssel als Pivotelement wählt, führt Quicksort auf bereits sortierten Arrays also quadratisch viele Schlüsselvergleiche aus.

Anzahl bewegter Schlüssel Wir haben soeben gesehen, dass Quicksort im schlechtesten Fall quadratisch viele Schlüsselvergleiche vornimmt, aber wie sieht es mit der Anzahl bewegter Schlüssel aus? Die Analyse der Schlüsselbewegungen im besten Fall bleibt als Übung dem Leser vorbehalten. Für die Analyse der Schlüsselbewegungen im schlechtesten Fall machen wir zunächst eine entscheidende Beobachtung:

Beobachtung

Ist ein Aufteilungsschritt vorbei, dann wurden höchstens so viele Schlüssel vertauscht wie die kürzere der beiden noch zu sortierenden Teilfolgen lang ist (siehe Abbildung 2.11)! Enthält die Eingabe des Aufteilungsschritts also n Schlüssel, dann werden im allgemeinen höchstens $n/2$ viele Vertauschungen vorgenommen. Allerdings führen die rekursiven Aufrufe weitere Vertauschungen durch, was die Analyse zunächst kompliziert erscheinen lässt. Daher bedienen wir uns eines Tricks:

Trick

Für jede durchgeführte Vertauschung nehmen wir an, dass der Schlüssel, der im späteren kürzeren Teil landet, eine Münze zahlt. Der andere an der Vertauschung beteiligte Schlüssel zahlt nichts. Die Gesamtanzahl der eingezahlten Münzen entspricht dann exakt der Anzahl von Schlüsselvertauschungen.

Kosten pro Schlüssel

Wie oft kann ein einzelner Schlüssel zahlen? Dazu überlegen wir, dass ein Schlüssel nur dann zahlt, wenn er in der kürzeren der noch zu sortierenden Teilfolgen landet. Diese hat, wie wir soeben gesehen haben, maximal halb so viele Schlüssel wie die ursprüngliche Folge. Jedes Mal, wenn ein Schlüssel zahlt, wird also die Länge des Bereichs, in dem er sich befindet, mindestens um die Hälfte kürzer. Da wir ursprünglich einen Bereich der Länge n hatten und nach $\log(n)$ vielen Halbierungen nur noch einen Bereich der Länge 1 haben (der trivialerweise sortiert ist), zahlt jeder Schlüssel also maximal $\log(n)$ viele Münzen. Da es insgesamt nur n Schlüssel gibt, werden also im schlechtesten Fall $\mathcal{O}(n \log n)$ viele Schlüssel vertauscht.

Randomisiertes Quicksort Obwohl Quicksort im schlechtesten Fall eine quadratische Laufzeit hat, wird der Algorithmus in der Praxis häufig eingesetzt. Ein Grund dafür besteht darin, dass nur wenige Wahlen eines Pivotelements zu einer quadratischen Laufzeit führen. Um nun zu vermeiden, dass in jedem Aufteilungsschritt ein ungünstiges Pivotelement gewählt wird, ersetzt man die deterministische Wahl eines Pivotelements (z.B., stets den am weitesten rechts liegenden Schlüssel) durch eine zufällige Wahl. Das heisst, im ersten Schritt von Partition(A, l, r) wählen wir eine zufällige natürliche Zahl z aus, die jeden der Werte $\{l, l+1, \ldots, r-1, r\}$ *mit gleicher Wahrscheinlichkeit* $1/(r-l+1)$ annimmt, tauschen $A[z]$ und $A[r]$ und führen den Aufteilungsschritt wie bisher durch. Wie schlecht kann dies werden? Nun, im schlechtesten Fall ändert sich natürlich nichts, denn wir könnten nach wie vor jedes Mal zufällig ein extrem schlechtes Pivotelement wählen. Solch ein Fall ist aber sehr unwahrscheinlich. Wir werden jetzt zeigen, dass eine zufällige Wahl im Durchschnitt zu guten Ergebnissen führt.

Zufällige Wahl des Pivotelements

Erwartete Anzahl von Schlüsselvergleichen

Wir haben bereits im Vorfeld argumentiert, dass die Anzahl bewegter Schlüssel bei einer Eingabe der Länge n immer in $\mathcal{O}(n \log n)$ liegt. Also genügt es, die erwartete Anzahl $T(n)$ miteinander verglichener Schlüssel zu analysieren. Dazu überlegen wir zunächst, dass eine geschickte Implementierung des Aufteilungsschritts auf einem Array der Länge n nur $n-1$ viele Schlüssel miteinander vergleichen muss (das Pivotelement wird genau einmal mit jedem anderen Schlüssel verglichen). Befindet sich nach dem Aufteilungsschritt das Pivotelement auf der Position k, dann können wir erwarten, dass das Sortieren des linken Teilbereichs $T(k-1)$ viele Schlüssel miteinander vergleicht, und das Sortieren des rechten Teilbereichs $T(n-k)$ viele Schlüssel miteinander vergleicht. Da das Pivotelement zufällig gleichverteilt gewählt wurde, befindet es sich nach Ende des Aufteilungsschritts auf jeder der Positionen $\{1, \ldots, n\}$ mit Wahrscheinlichkeit $1/n$. Die erwartete Anzahl verglichener Schlüssel bei einer Eingabe der Länge n beträgt also

$$T(n) = (n-1) + \frac{1}{n} \sum_{k=1}^{n-1} \Big(T(k-1) + T(n-k) \Big), \quad T(0) = T(1) = 0. \tag{46}$$

Setzen wir nun einige Werte für n in obige Formel ein, kann man zur Vermutung gelangen, dass $T(n) \leq 4n \log n$ gilt (wobei wir $0 \log 0 := 0$ definieren). Dies wollen wir nun durch vollständige Induktion über n beweisen.

Induktionsanfang ($n = 0, n = 1$): Da sowohl $T(0) = 0 \leq 4 \cdot 0 \log 0$ als auch $T(1) = 0 \leq 4 \cdot 1 \cdot \log 1$ gelten, ist die Behauptung sowohl für $n = 0$ als auch für $n = 1$ korrekt.

Induktionshypothese: Angenommen, es gäbe ein $n \geq 2$ sodass $T(n') \leq 4n' \log(n')$ für alle n' mit $0 \leq n' \leq 2$ gilt.

Induktionsschritt $(n-1 \to n)$:

$$T(n) = \left(\frac{2}{n}\sum_{k=1}^{n-1} T(k)\right) + n - 1 \tag{47}$$

$$\overset{I.H.}{\leq} \left(\frac{2}{n}\sum_{k=1}^{n-1} 4k\log k\right) + n - 1 \tag{48}$$

$$\overset{(*)}{=} \frac{2}{n}\left(\sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq(\log n)-1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n}\right) + n - 1 \tag{49}$$

$$= \frac{8}{n}\left(\big(\log n - 1\big)\sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k\right) + n - 1 \tag{50}$$

$$= \frac{8}{n}\left(\log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2} k\right) + n - 1 \tag{51}$$

$$= \frac{8}{n}\left(\log n \cdot \frac{n(n-1)}{2} - \frac{n}{4}\Big(\frac{n}{2}+1\Big)\right) + n - 1 \tag{52}$$

$$= 8\left(\log n \cdot \frac{n-1}{2} - \frac{n}{8} - \frac{1}{4}\right) + n - 1 \tag{53}$$

$$= 4(n-1)\log n - n - 2 + n - 1 \tag{54}$$

$$= 4n\log n - 4\log n - 3 \leq 4n\log n. \quad \blacksquare \tag{55}$$

Quicksort vergleicht also im Erwartungswert nur $\mathcal{O}(n \log n)$ viele Schlüssel miteinander. Tatsächlich ist auch die Konstante 4 zu hoch. Eine detaillierte Analyse (die wir hier nicht machen) würde sogar $T(n) \leq 3/2n \log n$ ergeben.

Zusätzlich benötigter Speicher Wir hatten Quicksort ja ursprünglich hergeleitet, um keinen zusätzlichen Platz mehr zu benötigen. Wir speichern auch keine Teilfolgen mehr zwischen, aber dafür wird Extraplatz zur Verwaltung der rekursiven Aufrufe benötigt. Schon früher haben wir uns überlegt, dass Quicksort die Eingabefolge der Länge n im schlimmsten Fall in eine Folge der Länge $n-1$ (und eine weitere der Länge 0) aufteilt. Wird diese Aufteilung so fortgesetzt, werden nicht nur quadratisch viele Schlüssel miteinander verglichen, wir haben auch eine *Rekursionstiefe* von n (denn der Aufruf auf der Eingabe der Länge n erzeugt einen Aufruf auf einer Eingabe der Länge $n-1$, diese erzeugt einen Aufruf auf einer Eingabe der Länge $n-2$, usw). Da die Verwaltung von jedem dieser rekursiven Aufrufe mindestens konstant viel Speicher braucht, benötigen wir im schlechtesten Fall insgesamt $\Omega(n)$ viel Speicher.

NAIVE IMPLEMENTIERUNG

Das Problem tritt nur dann auf, wenn wir die grössere der beiden Teilfolgen rekursiv sortieren. Würde hingegen nur die kleinere rekursiv sortiert, dann wären wir nach spätestens $\mathcal{O}(\log n)$ vielen rekursiven Aufrufen fertig, da sich die Grösse des entsprechenden Bereichs mindestens halbiert. Also tun wir exakt dies: Wir führen die Rekursion nur auf dem kürzeren Teil durch. Statt nun den längeren Teil rekursiv zu sortieren, führen wir den Aufteilungsschritt erneut auf dem längeren Teil durch, und sortieren dort wieder nur den kleineren Teil. Auf die gleiche Art verfahren wir so weiter, bis der verbleibende Bereich nur noch ein oder zwei Elemente umfasst.

REKURSIONSTIEFE BESCHRÄNKEN

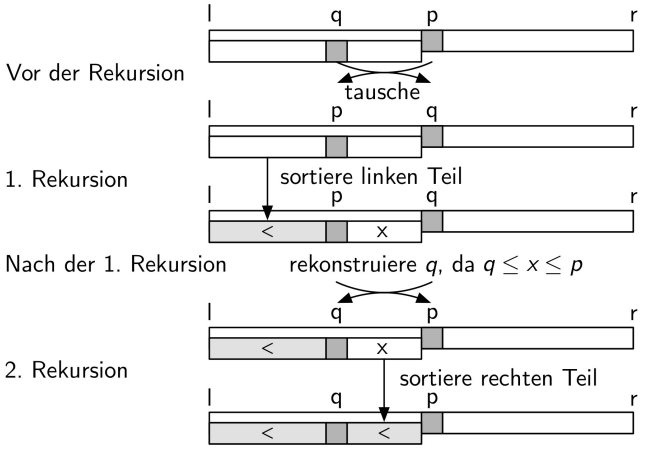


Abb. 2.12 Quicksort mit konstantem Extraplatz

Diese Modifikation sorgt also für eine Rekursionstiefe von $\mathcal{O}(\log n)$, und damit ist der benutzte Speicher in der gleichen Grössenordnung.

Konstanter Extraplatz

Für Jahrzehnte war es unklar, ob es noch besser geht, vielleicht sogar mit nur konstantem Extraplatz. Dies geht in der Tat mit folgender Idee. Bevor wir in den linken Teil der Rekursion absteigen, müssen wir uns das Pivotelement p irgendwie merken. Doch wie soll das gehen ohne Extraplatz? Wir merken uns die Grenze p, die den als zweites zu sortierenden Teilbereich markiert, mit Hilfe von q, dem Pivotelement des Bereichs links von p. Statt die Grenze p explizit als Funktionsargument auf den Stack zu kopieren, "verstecken" wir sie an der Position, an der sonst das Pivotelement q steht. Ist der linke Teil sortiert, so können wir die Länge des rechten Teils rekonstruieren, indem wir einfach nach rechts gehen, bis wir einen Schlüssel finden, der grösser als p ist. Wir vertauschen q und p und fahren mit dem Teil zwischen q und p fort. Danach wissen wir, dass der ganze Array links von p sortiert ist und wir können mit dem Teil rechts von p fortfahren (siehe Abbildung 2.12).

Ganz so einfach ist es in der konkreten Implementierung dann aber doch nicht, denn wir müssen diesen "Versteck-Schritt" ja rekursiv mehrfach durchführen. Dabei müssen wir sicherstellen, dass wir die Kette der versteckten Pivots der Reihe nach zurücktauschen können. Wenn wir das gleiche p einfach immer weiter hinunterschieben, geht das nicht. Aber man kann das zum Beispiel erreichen, wenn man nicht den Pivot versteckt, sondern das Element rechts vom Pivot. Man überlege sich die Details als Hausaufgabe. Obwohl diese modifizierte Variante von Quicksort mit $\mathcal{O}(1)$ Extraplatz auskommt, wird sie in der Praxis nicht verwendet, da der zusätzlich betriebene Aufwand für reale Anwendungen zu gross ist, und $\mathcal{O}(\log n)$ Extraplatz einerseits völlig akzeptabel und andererseits auch leicht erreichbar ist.

2.7 Eine untere Schranke für vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren haben gemeinsam, dass sie im schlechtesten Fall mindestens $\Omega(n \log n)$ viele Schlüsselvergleiche durchführen. Wir werden im Folgenden nur sog. *vergleichsbasierte* Sortierverfahren anschauen, die allein durch Schlüsselvergleiche und -vertauschungen sortieren, nicht aber durch Rechnungen mit Schlüsseln oder ähnlichem. Alle bisher vorgestellten Sortierverfahren (also Bubble-

Vergleichsbasiertes Sortieren

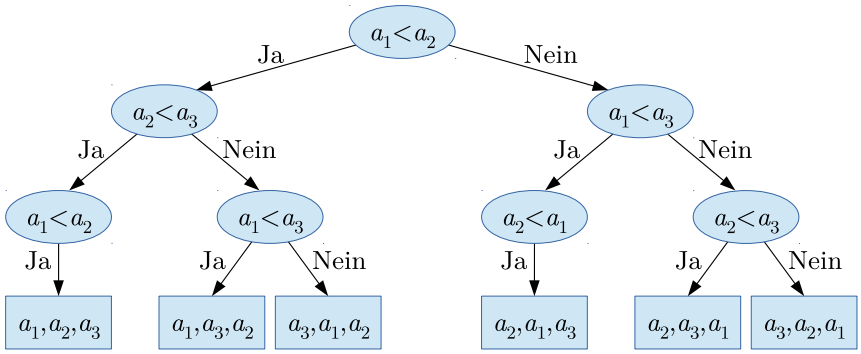


Abb. 2.13 Der Ablaufbaum für Bubblesort auf einem Array mit drei Schlüsseln. Jeder innere Knoten (rund) steht für den Vergleich zweier Schlüssel, jedes Blatt (eckig) für eine ermittelte sortierte Reihenfolge. Da die Höhe des Baums 4 ist, fallen im schlechtesten Fall genau drei Schlüsselvergleiche an.

sort, Sortieren durch Auswahl, Sortieren durch Einfügen, Heapsort, Mergesort und Quicksort) sind vergleichsbasiert. Wir überlegen uns nun, dass jedes vergleichsbasierte Sortierverfahren im schlechtesten Fall mindestens $\Omega(n \log n)$ viele Schlüssel miteinander vergleichen muss.

Schlechtester Fall

Wie zum Beweis der unteren Schranke für die Suche auf sortierten Arrays überlegen wir zunächst, dass jedes vergleichsbasierte Sortierverfahren als Entscheidungsbaum visualisiert werden kann (siehe Abbildung 2.13), in dem jeder innere Knoten einen Vergleich und jedes Blatt für eine ermittelte sortierte Reihenfolge steht. Die Anzahl der vom Algorithmus im schlechtesten Fall verglichenen Schlüsselpaare entspricht der Höhe des Baums (abzüglich 1). Da ein korrektes Sortierverfahren *jede* mögliche sortierte Reihenfolge korrekt erkennen muss, benötigt der Baum mindestens $n!$ viele Knoten (denn genau so viele Reihenfolgen gibt es). Da keine weiteren Vergleiche anfallen, sobald die Reihenfolge feststeht, braucht der Baum sogar $n!$ viele Blätter. Ein binärer Baum mit $n!$ vielen Blättern besitzt, wie früher gesehen, mindestens Höhe $\log(n!)$, was in $\Omega(n \log n)$ liegt (zu sehen z.B. mit der früher bewiesenen Abschätzung $\ln(n!) \geq n \ln n - n$). Jedes vergleichsbasierte Sortierverfahren muss also im schlechtesten Fall mindestens $\Omega(n \log n)$ viele Schlüssel miteinander vergleichen.

Durchschnittlicher Fall

Im schlechtesten Fall können wir also nicht besser werden. Vielleicht könnten wir aber ein Sortierverfahren entwerfen, das durchschnittlich weniger als $\mathcal{O}(n \log n)$ viele Schlüsselvergleiche durchführt. Wir werden nun zeigen, dass dies ebenfalls nicht möglich ist. Da jedes Blatt des Entscheidungsbaums eine mögliche sortierte Reihenfolge enthält und die Tiefe des Blatts die Anzahl der vom Algorithmus verglichenen Schlüsselpaare angibt, analysieren wir die *durchschnittliche* Tiefe eines Blatts. Sei T_n ein beliebiger Entscheidungsbaum mit n Blättern und $m(T_n)$ die mittlere Tiefe eines Blatts in T_n. Wir zeigen nun durch Widerspruch, dass stets $m(T_n) \geq \log n$ gilt. Angenommen, es gäbe einen Baum, für den die Aussage nicht gilt. Unter allen solchen Bäumen wählen wir einen mit minimaler Blattanzahl b, d.h. es gelte $m(T_b) < \log b$. Der Baum T_b besteht aus einer Wurzel mit einem linken Teilbaum T_{b_l} mit b_l Blättern und einem rechten Teilbaum T_{b_r} mit b_r Blättern (siehe Abbildung 2.14). O.B.d.A. nehmen wir an, dass sowohl T_{b_l} als auch T_{b_r} mindestens ein Blatt haben (man überlege als Hausaufgabe, warum wir dies annehmen können). Damit liegt jedes Blatt von T entweder in T_{b_l} oder in T_{b_r}, und wir haben $b = b_l + b_r$. Ausserdem sind sowohl b_l als auch b_r strikt kleiner als b. Da b die kleinstmögliche

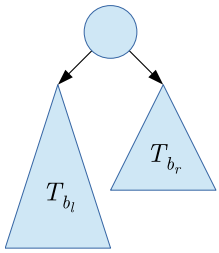


Abb. 2.14 Der Aufbau des Baums T_b aus einer Wurzel mit einem linken Teilbaum T_{b_l} und einem rechten Teilbaum T_{b_r}. Der linke Teilbaum hat b_l Blätter, der rechte b_r Blätter.

Anzahl von Blättern war, deren durchschnittliche Höhe strikt kleiner als $\log b$ ist, wissen wir, dass $m(T_{b_l}) \geq \log b_l$ und $m(T_{b_r}) \geq \log b_r$ gelten. Die durchschnittliche Tiefe eines Blatts in T_{b_l} und in T_{b_r} ist in T_b um 1 höher. Für die mittlere Tiefe eines Blatts in T_b erhalten wir damit

$$m(T_b) \geq \frac{b_l}{b}\big(\log b_l + 1\big) + \frac{b_r}{b}\big(\log b_r + 1\big) \tag{56}$$

$$= \Big(b_l\big(\log b_l + 1\big) + b_r\big(\log b_r + 1\big)\Big) \cdot \frac{1}{b} \tag{57}$$

$$= \Big(b_l \log b_l + b_r \log b_r + b\Big) \cdot \frac{1}{b} \tag{58}$$

$$\geq \frac{b}{2}\Big(\log \frac{b}{2} + 1\Big) \cdot 2 \cdot \frac{1}{b} = \log b, \tag{59}$$

was ein Widerspruch zur Annahme $m(T_b) < \log b$ ist (die letzte Ungleichung folgt aus der Tatsache, dass $x \log x$ konvex ist, damit ist $y \log y + z \log z \geq (y+z) \log \frac{y+z}{2}$ für alle $y, z \in \mathbb{R}^+$). Damit ist die durchschnittliche Tiefe eines Blatts in jedem Entscheidungsbaum mit n Blättern mindestens $\log(n)$. Da dies der durchschnittlichen Anzahl verglichener Schlüssel entspricht und der Entscheidungsbaum jedes vergleichsbasierten Sortierverfahrens mindestens $n!$ viele Blätter besitzt, benötigt jedes solche Verfahren auch im Durchschnitt $\log(n!) \in \Omega(n \log n)$ viele Schlüsselvergleiche.

Kapitel 3

Dynamische Programmierung

3.1 Einleitung

Schon früher haben wir mehrfach gesehen, dass Induktion beim systematischen Lösen von Problemen hilfreich sein kann. Es gibt viele Möglichkeiten, Probleme induktiv zu lösen, und wir wollen jetzt mit der sog. *Dynamischen Programmierung* eine weitere kennenlernen. Als einleitendes Beispiel betrachten wir einen Algorithmus zur Berechnung der sog. *Fibonacci-Zahlen*. Diese sind rekursiv als

Fibonacci-Zahlen

$$F_1 := 1, \quad F_2 := 1, \quad F_n := F_{n-1} + F_{n-2} \text{ für } n \geq 3 \tag{60}$$

definiert. Offenbar sind also $F_1 = F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, usw. Um nun für ein gegebenes $n \in \mathbb{N}$ die n-te Fibonacci-Zahl zu berechnen, könnten wir den folgenden rekursiven Algorithmus benutzen:

Rekursive Berechnung

FIBONACCI-RECURSIVE(n)

1 **if** $n \leq 2$ **then** $f \leftarrow 1$
2 **else** $f \leftarrow$ FIBONACCI-RECURSIVE($n-1$) + FIBONACCI-RECURSIVE($n-2$)
3 **return** f

Laufzeit

Sei nun $T(n)$ die Anzahl der von FIBONACCI-RECURSIVE(n) ausgeführten Operationen. Für $n = 1$ und $n = 2$ ist der Aufwand nur konstant, folglich ist $T(1) = \Theta(1)$. Für $n \geq 3$ haben wir einen rekursiven Aufruf für $n-2$, einen für $n-1$, und zusätzlich wird noch konstante Extrazeit zur Addition der Zahlen benötigt. Folglich ist $T(n) = T(n-2) + T(n-1) + c$ für eine geeignete Konstante c. Nun gilt

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c' \tag{61}$$

für eine geeignete Konstante c'. Der Algorithmus zur rekursiven Berechnung der n-ten Fibonacci-Zahl ist also exponentiell in n!

Der Grund für diese schlechte Laufzeit ist, dass manche identischen Teilprobleme wieder und wieder gelöst werden (siehe Abbildung 3.1). Um dieses Problem zu beheben, wenden wir einen einfachen Trick an, der unter dem Namen *Memoization* bekannt ist: Wir speichern alle Zwischenergebnisse (also alle Lösungen bereits gelöster Teilprobleme) und prüfen vor jedem Aufruf der Methode, ob das entsprechende Teilergebnis früher schon einmal berechnet wurde. Falls ja, dann geben wir einfach den gespeicherten Wert zurück. Nur wenn das Ergebnis noch nie berechnet wurde, berechnen wir es wie gehabt. Bevor wir es zurückgeben, speichern wir es, damit es ggf. später erneut verwendet werden kann.

Memoization

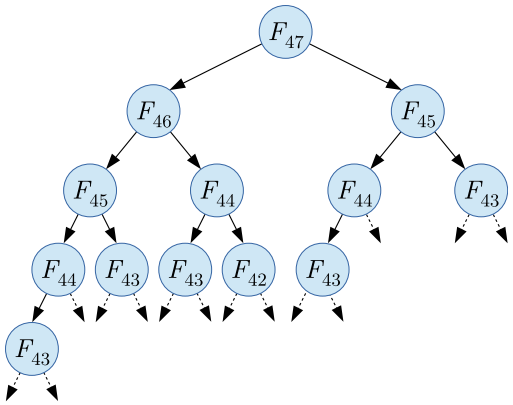


Abb. 3.1 Die obersten Ebenen des Rekursionsbaums, wenn FIBONACCI-RECURSIVE für $n = 47$ aufgerufen wird. Zur Berechnung von F_{47} müssen F_{46} und F_{45} berechnet werden, zur Berechnung von F_{46} entsprechend F_{45} und F_{44} usw. Viele Zahlen werden mehrfach berechnet. Man sieht z.B., dass fünf redundante Aufrufe zur Berechnung von F_{43} stattfinden!

Wir modifizieren nun den soeben gezeigten rekursiven Algorithmus, indem wir ein Array memo verwenden, das von 1 bis n indiziert ist, und das an Position i die i-te Fibonacci-Zahl F_i speichert, falls sie schonmal berechnet wurde, und **null** ansonsten. Damit erhalten wir den folgenden Algorithmus.

LÖSUNG MIT MEMOIZATION

FIBONACCI-MEMOIZATION(n)

```
1 if n-te Fibonacci-Zahl bereits berechnet then
2     f ← memo[n]                          ▷ Benutze gespeicherten Wert
3 else                                     ▷ Berechne n-te Fibonacci-Zahl
4     if n ≤ 2 then f ← 1
5     else f ← FIBONACCI-MEMOIZATION(n − 1)+
                FIBONACCI-MEMOIZATION(n − 2)
6     memo[n] ← f                          ▷ Speichere berechneten Wert
7 return f
```

LAUFZEIT

Auf den ersten Blick hat sich nicht viel verändert. Wir beobachten jetzt aber folgendes: Wird in Schritt 5 die Summe berechnet, dann sind nach dem Aufruf von FIBONACCI-MEMOIZATION($n-1$) sowohl F_{n-1} als auch F_{n-2} bereits vorberechnet. Der zweite Summand FIBONACCI-MEMOIZATION($n-2$) kann also in Zeit $\mathcal{O}(1)$ ausgerechnet werden. Abbildung 3.2 zeigt einen Teil des Rekursionsbaums für die Eingabe $n = 47$. Der Aufwand $T(n)$ für die Berechnung der n-ten Fibonacci-Zahl ist also konstant, falls sie früher schon einmal berechnet wurde, und beträgt ansonsten lediglich $T(n-1)+\mathcal{O}(1)$ (wobei $T(1)$ ebenfalls konstant ist). Lösen wir diese Rekurrenz auf, erhalten wir $T(n) \in \mathcal{O}(n)$, was eine beträchtliche Verbesserung gegenüber dem naiven Algorithmus darstellt. Wir brauchen lediglich zusätzlichen Speicher in der Grössenordnung von $\Theta(n)$. Allerdings benötigte auch der naive Algorithmus $\Theta(n)$ viel Speicher zur Verwaltung der rekursiven Aufrufe.

Anschaulich können wir uns vorstellen, dass wir die Knoten des Baums von oben nach unten berechnen, bis wir an den Blättern ankommen, die den Elementarfällen entsprechen oder bereits berechnete Ergebnisse speichern. Ein solches Vorgehen wird daher auch *Top-Down-Ansatz* genannt.

TOP-DOWN

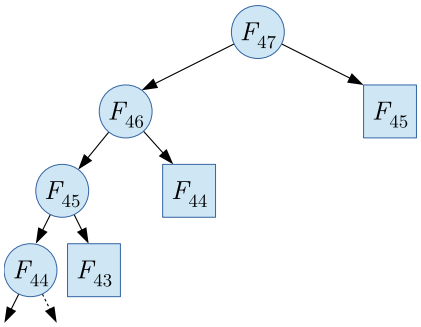


Abb. 3.2 Die obersten Ebenen des Rekursionsbaums, wenn FIBONACCI-MEMOIZATION für $n = 47$ aufgerufen wird. Wie vorher müssen zur Berechnung von F_{47} sowohl F_{46} und F_{45} berechnet werden, zur Berechnung von F_{46} entsprechend F_{45} und F_{44} usw. Im Gegensatz zum naiven Algorithmus sind jetzt aber die entsprechenden Teilergebnisse beim zweiten Aufruf bereits gespeichert und müssen daher nicht erneut berechnet werden. Rund dargestellte Knoten enthalten Werte, die berechnet werden müssen, eckig dargestellte Knoten beinhalten Werte, die bereits früher berechnet wurden.

Wir könnten auch umgekehrt vorgehen und die Werte zunächst für die Elementarfälle F_1 und F_2 explizit definieren und aufbauend auf diesen Werten $F_3, F_4, \ldots$ berechnen. Ein solches Vorgehen nennt man *Bottom-Up-Ansatz*, und wir sprechen dann auch von *dynamischer Programmierung*. Anschaulich gesprochen berechnen wir also den Baum von unten nach oben. Zur Berechnung der n-ten Fibonacci-Zahl ergibt sich dann der folgende Algorithmus.

BOTTOM-UP DYNAMISCHE PROGRAMMIERUNG

FIBONACCI-DYNAMIC-PROGRAM(n)

DYNAMISCHES PROGRAMM

```
1 F[1] ← 1
2 F[2] ← 1
3 for i ← 3, ..., n do
4       F[i] ← F[i − 1] + F[i − 2]
5 return F[n]
```

Obwohl dieser Bottom-Up-Algorithmus den gleichen asymptotischen Zeit- und Platzbedarf wie der zuvor vorgestellte Top-Down-Algorithmus besitzt, hat er dennoch einen grossen Vorteil: Er ist wesentlich kompakter und leichter verständlich.

VORTEIL

Dynamische Programmierung: Vorgehen Wenn wir Probleme mit dynamischer Programmierung lösen, verwalten wir immer eine *DP-Tabelle* (auch *Tableau* genannt), welche Informationen über bereits gelöste Teilprobleme enthält. Oftmals sind dies die Werte optimaler Lösungen der entsprechenden Teilprobleme, aber auch Informationen über die Lösung selbst könnten dort gespeichert werden. Zu Beginn werden all die Einträge gefüllt, die den Elementar- bzw. Randfällen entsprechen. Wie wir für die später diskutierten Beispiele noch sehen werden, sind diese oftmals sehr einfach berechenbar. Danach werden die weiteren Einträge der Tabelle sukzessive ausgefüllt. Dabei wird im Allgemeinen auf weitere, bereits früher berechnete Einträge der Tabelle zugegriffen. Da diese weiteren Einträge *vor* der Berechnung eines Eintrags bereits fertig berechnet sein müssen, muss man also im Vorfeld die Reihenfolge spezifizieren, in der die Tabelle gefüllt werden soll. Nachdem die Tabelle

DP-TABELLE / TABLEAU

Systematisches Vorgehen

komplett berechnet ist, muss noch die Lösung für das ursprüngliche Problem abgelesen werden. Bei der Lösung eines Problems mittels dynamischer Programmierung sollte man also stets die folgenden vier Aspekte berücksichtigen:

1) *Definition der DP-Tabelle:* Welche Dimensionen hat die Tabelle? Was ist die Bedeutung jedes Eintrags?

   Beispiel: Wir benutzen eine Tabelle T der Grösse $1 \times n$, wobei der i-te Eintrag die i-te berechnete Fibonacci-Zahl enthält.

2) *Berechnung eines Eintrags:* Wie berechnet sich ein Eintrag aus den Werten von anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?

   Beispiel: Für $i = 1$ und $i = 2$ setzen wir $T[i] \leftarrow 1$. Für ein allgemeines $i \geq 3$ setzen wir $T[i] \leftarrow T[i-1] + T[i-2]$.

3) *Berechnungsreihenfolge*: In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden?

   Beispiel: Die Einträge $T[i]$ werden mit aufsteigendem i berechnet. Wir berechnen also zuerst $T[1]$, dann $T[2]$, dann $T[3]$, usw.

4) *Auslesen der Lösung:* Wie lässt sich die Lösung am Ende aus der Tabelle auslesen?

   Beispiel: Die zu berechnende n-te Fibonacci-Zahl ist am Ende im Eintrag $T[n]$ gespeichert.

Die Laufzeit eines dynamischen Programms berechnet sich oftmals einfach aus der Grösse der Tabelle multipliziert mit dem Aufwand, jeden Eintrag zu berechnen. Dies ist auch der Fall für das soeben diskutierte Beispiel: Die Tabelle hat n Einträge und die Berechnung eines jeden Eintrags erfordert lediglich konstante Zeit, folglich ist die Gesamtlaufzeit in $\mathcal{O}(n)$. In dynamischen Programmen für andere Probleme kann aber auch der Aufwand zur Berechnung einer geeigneten Reihenfolge oder zum Auslesen der Lösung die Gesamtlaufzeit dominieren.

Um ein Problem mit dynamischer Programmierung zu lösen, sollte man sich zuerst überlegen, wie man das Problem in geeignete Teilprobleme mit einer identischen Struktur zerlegen kann. Um sich dies klarzumachen, kann es hilfreich sein, das Problem zunächst rekursiv zu lösen. Danach beginnt die eigentliche Arbeit: Man muss sich nämlich überlegen, wie man die Ergebnisse der rekursiven Aufrufe geeignet in der DP-Tabelle speichern kann. Um das eigentliche dynamische Programm aufzustellen, sollte man dann die zuvor genannten vier Aspekte berücksichtigen. Wir werden nun für einige Problem anschauen, wie sie mit dynamischer Programmierung gelöst werden können.

3.2 Längste aufsteigende Teilfolge

Motivation Wir untersuchen jetzt ein Problem, das in ähnlicher Form beim Design von VLSI-Chips auftritt. In diesem sind zwei gegenüberliegende Reihen von punktförmigen Anschlüssen gegeben. Die Anschlüsse der ersten Reihe sind von links nach rechts mit den Zahlen $1, \ldots, n$ nummeriert. Die Anschlüsse der zweiten Reihe

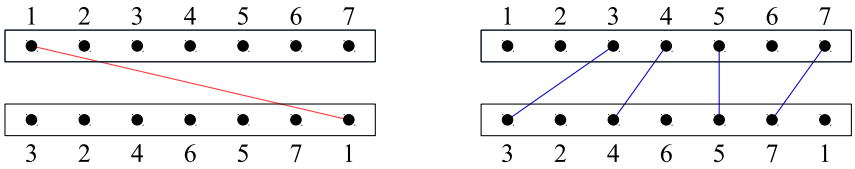


Abb. 3.3 Zwei mögliche Lösungen des Anschlussverbindungsproblems. Auf der linken Seite ist eine gültige Lösung dargestellt, in der die Anschlüsse mit der Nummer 1 korrekt miteinander verbunden sind. Jede weitere Verbindung zwischen zwei anderen passenden Anschlüssen (also z.B. von 2 oben nach 2 unten) würde die rote Verbindung kreuzen, und die entstehende Lösung wäre ungültig. Auf der rechte Seite ist eine gültige Lösung dargestellt, die so viele Anschlüsse wie möglich miteinander verbindet.

tragen ebenfalls die Nummern $1, \ldots, n$, aber diese Zahlen sind in beliebiger Reihenfolge angeordnet (vgl. Abbildung 3.3). Wir wollen nun so viele passende Anschlüsse (d.h., Anschluss i in der ersten Reihe mit Anschluss i in der zweiten Reihe) wie möglich miteinander verbinden, dabei aber vermeiden, dass zwei Verbindungen sich kreuzen (denn wenn zwei elektrische Leitungen sich kreuzen, gibt es einen Kurzschluss).

Modellierung Bevor wir uns überlegen, wie dieses Problem gelöst werden kann, müssen wir es zunächst etwas formalisieren. Dazu nehmen wir an, dass die Anschlüsse in der zweiten Reihe von links nach rechts mit den Zahlen $a_1, \ldots, a_n$ nummeriert sind. Um die Kreuzungsfreiheit zu modellieren, machen wir folgende Beobachtung: In jeder gültigen, kreuzungsfreien Lösung für das Anschlussverbindungsproblem sind die Nummern der verbundenen Anschlüsse in der zweiten Reihe von links nach rechts aufsteigend geordnet. Wir müssen also in der Sequenz $A = (a_1, \ldots, a_n)$ eine längste aufsteigende Teilfolge suchen. Dies ist eine Teilsequenz von A, deren Elemente von links nach rechts aufsteigend sortiert sind. Ist zum Beispiel die Sequenz $A = (1, 2, 4, 3)$ gegeben, dann sind sowohl $(1, 2, 4)$ als auch $(1, 2, 3)$ längste aufsteigende Teilfolgen. Auch (1), (2), (4), (3), $(1, 2)$, $(1, 4)$, $(1, 3)$, $(2, 4)$ und $(2, 3)$ sind aufsteigende Teilfolgen von A, sie sind aber nicht längstmöglichst. Die Teilfolge $(2, 4, 3)$ dagegen ist nicht aufsteigend, da $4 > 3$ gilt.

Längste aufsteigende Teilfolge

Im Folgenden verallgemeinern wir das Problem ein wenig, indem wir auch Eingaben zulassen, die andere als die Zahlen $\{1, \ldots, n\}$ enthalten. Auch erlauben wir, dass manche Zahlen in der Eingabe mehrfach vorkommen. Gesucht wird dann eine *strikt aufsteigende* Teilfolge. Eine längste aufsteigende Teilfolge der Eingabe $(2, 3, 3, 3, 5, 1)$ ist etwa $(2, 3, 5)$ (und nicht $(2, 3, 3, 3, 5)$).

Wir wollen nun überlegen, wie man das Problem zur Berechnung einer längsten gemeinsamen Teilfolge (im Folgenden verkürzend *LAT* genannt) auf geeignete kleinere Probleme zurückführen kann. Dazu werden wir einige mögliche Ideen untersuchen.

Entwurf 1 Angenommen, wir hätten bereits eine LAT für die ersten k Elemente berechnet, und wir wollen nun eine LAT für die ersten $k + 1$ Elemente berechnen. Dazu betrachten wir das Element a_{k+1}. Falls dieses an die vorher berechnete LAT passt, dann sind wir fertig. Was können wir aber tun, falls a_{k+1} nicht an diese LAT passt? Ein Problem bei diesem Entwurf besteht darin, dass die LAT im allgemeinen nicht eindeutig ist. Angenommen, es wäre $A = (1, 2, 5, 3, 4)$ gegeben, und wir kennen

bereits die LAT der ersten $k = 3$ Elemente (also für die Teilsequenz $(1, 2, 5)$). Diese LAT ist natürlich $(1, 2, 5)$. Das Element $a_{k+1} = 3$ kann nun nicht angehängt werden. Dies wäre aber wichtig, um später die LAT $(1, 2, 3, 4)$ für die ersten fünf Elemente (also für die Sequenz A selbst) zu berechnen.

Entwurf 2 Die vorige Annahme, nur *irgendeine* LAT für die ersten k Elemente zu kennen, war offenbar zu schwach. Nehmen wir also nun an, wir hätten bereits *alle* LATs für die ersten k Elemente berechnet. Um nun alle LATs für die ersten $k + 1$ Elemente zu berechnen, können wir einfach prüfen, an welche LATs das Element a_{k+1} passt, und die Folgen entsprechend erweitern. Dies ist aber nicht unproblematisch, wie das Beispiel $A = (1, 2, 5, 4, 3)$ zeigt. Angenommen, wir kennen bereits alle LATs für die ersten $k = 4$ Elemente. Diese LATs sind $(1, 2, 5)$ und $(1, 2, 4)$. Das Element $a_{k+1} = 3$ passt an keine der bestehenden LATs. Wir beobachten aber, dass auch $(1, 2, 3)$ eine LAT ist. Folglich müsste die Liste der LATs verlängert werden, und wir müssten uns nicht nur alle LATs maximaler Länge l merken, sondern auch noch alle LATs der Länge $l - 1$, $l - 2$, usw. Dies ist definitiv zu viel, daher ist der Entwurf nicht praktikabel.

Entwurf 3 Die Annahme im ersten Entwurf war zu schwach, die Annahme im zweiten Entwurf war zu stark. Das soeben gezeigte Beispiel, in dem die LATs der ersten vier Elemente genau $(1, 2, 5)$ und $(1, 2, 4)$ waren, lässt jedoch vermuten, dass es möglicherweise gar nicht erforderlich ist, sich alle LATs zu merken. Stattdessen genügt es vielleicht auch, sich diejenige LAT zu merken, die *mit dem kleinsten Element endet*, denn kann diese LAT nicht erweitert werden, dann kann es auch keine andere. Was können wir aber in solch einem Fall tun? Angenommen, eine solche LAT der ersten k Elemente hat Länge l, das Element a_{k+1} passt nicht an diese LAT (da a_{k+1} kleiner als das letzte Element der LAT ist), aber an die ersten $k - 1$ Elemente der LAT. Ein Beispiel wäre die Eingabe $A = (1, 2, 5, 4, 3)$, wo die entsprechende LAT der ersten $k = 4$ Elemente genau $(1, 2, 4)$ wäre, die nicht durch $a_{k+1} = 3$ erweitert werden kann. Wird dann das $(k + 1)$-te Element hinzugenommen, ist die LAT nicht mehr $(1, 2, 4)$, sondern $(1, 2, 3)$. Sind wir also fertig?

IDEE **Entwurf 4 (Final)** Wir müssen noch berücksichtigen, dass u.U. auch LATs kürzerer Länge aktualisiert werden müssen. Dies kann im späteren Verlauf des Algorithmus noch wichtig werden, wie z.B. die Eingabe $A = (1, 1000, 1001, 2, 3, 4, \ldots, 999)$ zeigt. Die einzige LAT der ersten $k = 3$ Elemente ist $(1, 1000, 1001)$. Wird jetzt das Element $a_{k+1} = 2$ gefunden, dann kann dieses nur an die 1 angehängt werden. Dies wird im späteren Verlauf tatsächlich noch gebraucht, denn sonst könnte die LAT $(1, 2, 3, \ldots, 999)$ nicht berechnet werden.

Wir merken uns also für jede mögliche Länge $l \in \{1, \ldots, k\}$ diejenige LAT der Länge l, die mit dem kleinsten Element endet. Wird dann ein neues Element a_{k+1} betrachtet, dann suchen wir die längste LAT, die noch durch a_{k+1} erweiterbar ist. Sei l' die Länge dieser LAT. Wir hängen a_{k+1} an diese LAT an und speichern sie als diejenige LAT der Länge $l' + 1$, die mit dem kleinsten Element endet.

Dynamisches Programm Nachdem wir uns eine Lösungsidee bereits überlegt haben, wollen wir nun ein geeignetes dynamisches Programm formulieren. Dazu speichern wir keine vollständigen LATs zwischen, sondern merken uns für jede Länge l

lediglich das letzte Element der entsprechenden LAT. Man beachte, dass diese Elemente mit aufsteigender Länge l aufsteigend sortiert sind. Um später die LAT selbst rekonstruieren zu können, speichern wir zudem für jedes Element a_k, welches Element der Vorgänger in einer LAT ist, die in a_k endet.

Dynamisches Programm

1) *Definition der DP-Tabelle:* Wir benutzen eine Tabelle T, die von 0 bis n indiziert ist. Für $l > 0$ speichert $T[l]$ das letzte Element einer LAT der Länge l, deren letztes Element kleinstmöglich ist, falls eine solche existiert. Ansonsten ist $T[l] = +\infty$. $T[0]$ speichert stets $-\infty$. Zudem benutzen wir eine weitere Tabelle V, die von 1 bis n indiziert ist. Der Eintrag $V[i]$ speichert den Vorgänger von a_i in einer LAT, deren letztes Element a_i ist.

2) *Berechnung eines Eintrags:* Vor Beginn des dynamischen Programms muss die Tabelle T initialisiert werden. Dazu setzen wir $T[0] \leftarrow -\infty$ und $T[l] \leftarrow +\infty$ für alle Längen $l > 0$.

   Für jedes $k \in \{1, \dots, n\}$ wollen wir die längste LAT ermitteln, die um a_k erweitert werden kann. Da die Einträge in T aufsteigend sortiert sind, führen wir eine binäre Suche in T aus, um die Position l des am weitesten rechts stehenden Elements in T zu finden, das kleiner als a_k ist. Nach Terminierung der binären Suche gilt also $T[l] < a_k < T[l+1]$. Da $T[l]$ das kleinste letzte Element einer LAT der Länge l speichert, kann die entsprechende LAT durch a_k erweitert werden. Folglich setzen wir $T[l+1] \leftarrow a_k$. Als Vorgänger dieses Elements setzen wir $V[k] \leftarrow T[l]$ (denn $T[l]$ ist das letzte Element derjenigen LAT, an die das aktuelle Element a_k gehängt wurde).

3) *Berechnungsreihenfolge:* Die Einträge $T[i]$ und $V[i]$ werden mit aufsteigendem i berechnet. Wir berechnen also zuerst $T[0]$, dann $T[1]$ und $V[1]$, dann $T[2]$ und $V[2]$, usw.

4) *Auslesen der Lösung:* Wir rekonstruieren die Lösung rückwärts, indem wir die Eingabe A rückwärts durchlaufen und die Tabellen T und V benutzen. Zuerst wird T von rechts her durchlaufen, um die grösste Position l mit $T[l] < \infty$ zu finden. Dann ist l die Länge einer LAT, und $T[l]$ ist das letzte Element einer LAT. Wir geben also $T[l]$ aus und suchen nun den Index i_1 von $T[l]$ in A, indem A von rechts her durchlaufen wird. Nun geben wir den Vorgänger von $T[l]$ aus. Dieser ist in $V[i_1]$ gespeichert (daher musste der Index von $T[l]$ erst berechnet werden). Nun suchen wir nach dem Index i_2 von $V[i_1]$ in A, indem A von i_1 ausgehend weiter nach links durchlaufen wird. Dann geben wir $V[i_2]$ aus, suchen den Index i_3 mit $a_{i_3} = V[i_2]$, und verfahren auf diese Art weiter, bis alle Elemente der LAT rekonstruiert sind.

Laufzeit

Wie effizient ist dieser Algorithmus? In der Initialisierung werden genau $n+1 \in \Theta(n)$ viele Zuweisungen ausgeführt. Zur Berechnung des k-ten Eintrags wird zunächst eine binäre Suche auf den Positionen $\{1, \dots, k\}$ durchgeführt, danach werden nur noch konstant viele Zuweisungen vorgenommen. Der Gesamtaufwand zur Berechnung der Tabelle beträgt also

$$\sum_{k=1}^{n} \big(\log(k) + \mathcal{O}(1)\big) = \mathcal{O}(n) + \sum_{k=1}^{n} \log(k) = \Theta(n \log n). \tag{62}$$

Zur Rekonstruktion der Lösung muss im Wesentlichen das Array A einmal von rechts nach links durchlaufen werden. Darüber hinaus werden pro rekonstruiertem Element

nur noch konstant viele zusätzliche Operationen ausgeführt, folglich kann die Lösung in Zeit $\mathcal{O}(n)$ aus der Tabelle extrahiert werden. Die Gesamtlaufzeit des dynamischen Programms beträgt also $\Theta(n \log n) + \mathcal{O}(n) = \Theta(n \log n)$.

3.3 Längste gemeinsame Teilfolge

Problemformulierung Wir wollen nun ein ähnliches wie das soeben betrachtete Problem untersuchen, und auch für dieses überlegen, wie dynamische Programmierung zur Lösung benutzt werden kann.

Eine Teilfolge einer Zeichenkette Z ist eine Auswahl von Zeichen aus Z, von links nach rechts geschrieben. Die Zeichenkette `KUH` etwa hat acht Teilfolgen, nämlich die leere Teilfolge, die drei Teilfolgen `K`, `U` und `H` der Länge 1, die drei Teilfolgen `KU`, `UH` und `KH` der Länge 2, sowie die Teilfolge `KUH` der Länge 3. Dagegen sind z.B. `KB`, `UHU` und `HK` keine Teilfolgen von `KUH`.

PROBLEM-FORMULIERUNG

Die Eingabe unseres Problems besteht aus zwei Zeichenketten $A = (a_1, \ldots, a_m)$ und $B = (b_1, \ldots, b_n)$ mit jeweils m bzw. n Zeichen. Gesucht ist eine längste gemeinsame Teilfolge (LGT) von A und B. Die LGT von $A =$ `IGEL` und $B =$ `KATZE` ist also `E`, die LGT von $A =$ `RIND` und $B =$ `PFERD` ist `RD`, und die LGT von $A =$ `TIGER` und $B =$ `ZIEGE` ist `IGE`. Anwendung findet dieses Problem etwa in der Biologie, um die Ähnlichkeit von DNA-Sequenzen zu untersuchen.

Rekursive Lösung Wir überlegen zunächst, wie wir das Problem rekursiv lösen könnten. Man kann sich vorstellen, die Zeichenketten A und B so untereinander zu schreiben, dass man die LGT direkt sieht. Dazu müssen ggf. Leerzeichen eingeschoben werden. Für die Eingaben $A =$ `TIGER` und $B =$ `ZIEGE` erhalten wir z.B.

T	I	_	G	E	R
Z	I	E	G	E	_

Wie hilft uns dies nun zur rekursiven Lösung des Problems? Angenommen, wir könnten bereits für jedes $i \in \{0, \ldots, n\}$ und jedes $j \in \{0, \ldots, n\}$ (ausgenommen für $i = m$ und $j = n$ gleichzeitig) die Länge $L(i, j)$ der LGT von $A[1..i]$ und $B[1..j]$ berechnen (also die Länge der LGT der ersten i Zeichen von A und den ersten j Zeichen von B). Betrachten wir jetzt nur die letzten beiden Zeichen a_m von A und b_n von B, dann gibt es nur drei Möglichkeiten:

1) Die LGT von A und B entspricht der LGT von $A[1..m-1]$ und $B[1..n-1]$ (falls $a_m \neq b_n$) bzw. der LGT von $A[1..m-1]$ und $B[1..n-1]$ um a_m erweitert (falls $a_m = b_n$). Dann gilt offenbar $L(m, n) = L(m-1, n-1) + \delta_{mn}$, wobei $\delta_{mn} = 1$ ist, falls $a_m = b_n$ gilt, und $\delta_{mn} = 0$ ansonsten.

2) A wird um ein Leerzeichen erweitert, folglich wird b_n ignoriert. Es gilt also $L(m, n) = L(m, n-1)$.

3) B wird um ein Leerzeichen erweitert, folglich wird a_m ignoriert. Es gilt also $L(m, n) = L(m-1, n)$.

Da wir nicht wissen, welcher dieser drei Möglichkeiten maximale Länge hat, berechnen wir einfach rekursiv alle Möglichkeiten und setzen daher

$$L(m, n) \leftarrow \max\Big(L(m-1, n-1) + \delta_{mn},\ L(m, n-1),\ L(m-1, n)\Big), \tag{63}$$

L		Z	I	E	G	E
	0	0	0	0	0	0
T	0	0	0	0	0	0
I	0	0	1	1	1	1
G	0	0	1	1	2	2
E	0	0	1	2	2	3
R	0	0	1	2	2	3

Abb. 3.4 Die DP-Tabelle zur Berechnung einer längsten gemeinsamen Teilfolge für die Eingabe $A =$ TIGER und $B =$ ZIEGE. Zur Rekonstruktion der längsten gemeinsamen Teilfolge IGE werden die markierten Tabelleneinträge betrachtet.

und für die Randfälle legen wir explizit $L(0,\cdot) \leftarrow 0$ und $L(\cdot,0) \leftarrow 0$ fest (denn eine LGT einer Zeichenkette und einer leeren Zeichenkette umfasst 0 Zeichen).

Dynamisches Programm Die Lösung im vorigen Abschnitt kann direkt in ein dynamisches Programm überführt werden, indem einfach alle Zwischenergebnisse in einer zweidimensionalen Tabelle gespeichert werden.

Dynamisches Programm

1) *Definition der DP-Tabelle:* Wir benutzen eine zweidimensionale Tabelle L, die in der ersten Dimension von 0 bis m und in der zweiten Dimension von 0 bis n indiziert ist. Dabei gibt $L[i,j]$ die Länge einer LGT der Zeichenketten $(a_1,\ldots,a_i)$ und $(b_1,\ldots,b_j)$ an.

2) *Berechnung eines Eintrags:* Die Tabelle L wird initialisiert, indem $L[i,0] \leftarrow 0$ für alle $i \in \{0,\ldots,m\}$ und $L[0,j] \leftarrow 0$ für alle $j \in \{1,\ldots,n\}$ gesetzt werden.

   Zur Berechnung von $L[i,j]$ mit $i,j \geq 1$ setzen wir

$$L[i,j] \leftarrow \max\Big(L[i-1,j-1] + \delta_{ij},\, L[i,j-1],\, L[i-1,j]\Big),$$

   wobei $\delta_{ij} = 1$ für $a_i = b_j$ ist, und $\delta_{ij} = 0$ sonst.

3) *Berechnungsreihenfolge*: Zur Berechnung des Eintrags $L[i,j]$ müssen die Einträge $L[i-1,j-1]$, $L[i,j-1]$ und $L[i-1,j]$ bereits berechnet sein. Es gibt mehrere Berechnungsreihenfolgen, die dies garantieren. Wir können z.B. die Einträge $L[i,j]$ nach aufsteigendem i, und für gleiches i nach aufsteigendem j berechnen. Wir berechnen also zuerst $L[0,0]$, dann $L[0,1]$ bis $L[0,n]$, dann $L[1,0]$ bis $L[1,n]$, usw.

4) *Auslesen der Lösung:* Am Ende speichert $L[m,n]$ die Länge einer LGT von A und B. Um die LGT selbst zu rekonstruieren, laufen wir in der Tabelle zurück und geben immer dann das entsprechende Zeichen aus, wenn die Länge um 1 erhöht wurde. Konkret setzen wir initial $i \leftarrow m$ und $j \leftarrow n$. In jedem Schritt prüfen wir dann, ob $a_i = b_j$ und $L[i,j] = L[i-1,j-1] + 1$ gelten. Falls ja, dann geben wir a_i aus, setzen $i \leftarrow i-1$ und $j \leftarrow j-1$. Falls nein, dann prüfen wir, ob $L[i,j] = L[i-1,j]$ oder $L[i,j-1]$ gilt und verringern entsprechend i oder j um 1. Wir fahren auf die gleiche Art fort, bis entweder $i = 0$ oder $j = 0$ gilt. Danach haben wir die LGT (rückwärts gelesen) rekonstruiert.

   Schaut man sich den Berechnungsschritt genau an, dann wird man merken, dass es beim Auslesen der Lösung schon genügt, nur $a_i = b_j$ zu prüfen. In

diesem Fall gilt immer $L[i,j] = L[i-1,j-1]+1$, und wir können die Rekonstruktion von der Position $L[i-1,j-1]$ aus fortsetzen. Ansonsten fahren wir mit der Position $L[i,j-1]$ fort, falls $L[i,j] = L[i,j-1]$ gilt, und bei $L[i-1,j]$ sonst.

LAUFZEIT Die Laufzeit dieses dynamischen Programms lässt sich leicht ermitteln. Die DP-Tabelle hat $(m+1)\cdot(n+1)$ viele Einträge, und jeder Eintrag lässt sich durch eine konstante Anzahl von Zuweisungen und Vergleichen berechnen. Die benötigte Zeit zur Berechnung der Tabelle ist also in $\mathcal{O}(mn)$. Beim Auslesen der Lösung starten wir mit $i = m$ und $j = n$ und verringern in jedem Schritt entweder i oder j (oder sogar beide) um 1, bis $i = 0$ oder $j = 0$ (oder beides) gilt. Damit kann die (rückwärts gelesene) Lösung in Zeit $\mathcal{O}(m+n)$ ermittelt werden. Insgesamt ergibt sich damit also eine Laufzeit von $\mathcal{O}(mn)$.

3.4 Minimale Editierdistanz

Ein weiteres Problem, das gewisse Ähnlichkeiten mit der Berechnung einer längsten gemeinsamen Teilfolge hat, ist die Berechnung der *Editierdistanz* zweier gegebener Zeichenketten $A = (a_1,\ldots,a_m)$ und $B = (b_1,\ldots,b_n)$. Die sogenannten *Editieroperationen* für eine gegebene Zeichenkette sind

EDITIER-OPERATIONEN

- die Einfügung eines neuen Zeichens,
- die Löschung eines bestehenden Zeichens, sowie
- die Änderung eines bestehenden Zeichens.

Man könnte diese Operationen auch gewichten. So z.B. könnte man festlegen, dass die Einfügung des Zeichens `X` Kosten 24 und die Löschung des Zeichens `Y` Kosten 1 hat, aber so weit wollen wir jetzt nicht gehen. Stattdessen wollen wir einfach nur untersuchen, wie viele Editieroperationen mindestens nötig sind, um eine gegebene Zeichenkette A in eine Zeichenkette B zu überführen.

PROBLEM-FORMULIERUNG

BEISPIEL Wollen wir z.B. die Zeichenkette $A =$ `TIGER` in $B =$ `ZIEGE` überführen, können wir zuerst das `T` durch ein `Z` ersetzen, dann zwischen `I` und `G` ein `E` einfügen, und schliesslich den letzten Buchstaben `R` löschen. Die Editierdistanz von A und B ist also höchstens 3 (man kann leicht feststellen, dass man tatsächlich nicht mit weniger Editieroperationen auskommt).

IDEE Zur Lösung dieses Problems können wir ähnlich wie zur Berechnung einer längsten gemeinsamen Teilfolge vorgehen. Dazu benutzen wir wieder eine zweidimensionale DP-Tabelle E, die in der ersten Dimension von 0 bis m und in der zweiten Dimension von 0 bis n indiziert ist. Ein Eintrag $E[i,j]$ gibt die Editierdistanz der Teilzeichenketten $A' = (a_1,\ldots,a_i)$ und $B' = (b_1,\ldots,b_j)$ an. Diese berechnen wir als

$$E[i,j] \leftarrow \min\Big(E[i-1,j]+1,\, E[i,j-1]+1,\, E[i-1,j-1]+1-\delta_{ij}\Big) \qquad (64)$$

(mit $\delta_{ij} = 1$ genau dann, wenn $a_i = b_j$ ist, und $\delta_{ij} = 0$ sonst), denn betrachten wir nur die jeweils letzten Zeichen von A' und B', dann können wir entweder

- das letzte Zeichen von A' löschen ($E[i-1,j]+1$),
- B' ein Zeichen hinzufügen ($E[i,j-1]+1$), oder

E		Z	I	E	G	E
	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	3	3	2
R	5	5	4	4	4	3

Abb. 3.5 Die DP-Tabelle zur Berechnung der Editierdistanz der Zeichenketten $A =$ TIGER und $B =$ ZIEGE. Zur Rekonstruktion der benötigten Editieroperationen werden die markierten Tabelleneinträge betrachtet.

- a_i durch b_j ersetzen ($E[i-1, j-1]$ falls $a_i = b_j$ ist, und $E[i-1, j-1] + 1$ sonst).

Für die Randfälle setzen wir $E[i, 0] \leftarrow i$ für alle $i \in \{0, \ldots, m\}$ (denn die Editierdistanz zwischen einer Zeichenkette der Länge i und der leeren Zeichenkette beträgt exakt i), sowie $E[0, j] \leftarrow j$ für alle $j \in \{1, \ldots, n\}$ (denn die Editierdistanz zwischen einer leeren Zeichenkette und einer Zeichenkette der Länge j beträgt exakt j).

LAUFZEIT

Die Gesamtlaufzeit des Verfahrens beträgt $\mathcal{O}(mn)$. Da das Vorgehen ähnlich zur Berechnung einer längsten gemeinsamen Teilfolge ist, werden wir das dynamische Programm nicht genauer beschreiben. Daher sei dem Leser an dieser Stelle dringend empfohlen, die Details auszuarbeiten und sich insbesondere zu überlegen, wie man aus der ausgefüllten Tabelle die entsprechende Folge von Editieroperationen auslesen kann.

3.5 Matrixkettenmultiplikation

PROBLEM

Angenommen, wir haben n Matrizen $A_1, \ldots, A_n$ und wollen das Produkt

$$A_1 \times A_2 \times \cdots \times A_n \tag{65}$$

berechnen. Eine naive Vermutung ist, dass man einfach $P_1 := (A_1 \times A_2)$ berechnen sollte, dann $P_2 := (P_1 \times A_3)$, dann $P_3 := (P_2 \times A_4)$, usw. Dieses Vorgehen ist aber oftmals nicht sinnvoll, denn die Matrizenmultiplikation ist assoziativ. Das heisst also, wir können die Klammerung bei der Matrizenmultiplikation beliebig wählen. So zum Beispiel gilt

$$((A_1 \times ((A_2 \times A_3) \times A_4)) \times A_5) = ((A_1 \times A_2) \times A_3) \times (A_4 \times A_5). \tag{66}$$

Das Resultat ist zwar immer gleich, aber der Rechenaufwand kann sich je nach Klammerung ändern (siehe Abbildung 3.6). Die Multiplikation einer $(r \times s)$-Matrix A_1 mit einer $(s \times u)$-Matrix A_2 hat die Kosten $r \cdot s \cdot u$, und als Ergebnis erhalten wir eine $(r \times u)$-Matrix. Wir erhoffen uns nun, dass die Bestimmung einer optimalen Klammerung wesentlich schneller als die anschliessende Ausführung der Multiplikationen dauert. Für eine Bibliothek mathematischer Software kann es dann durchaus Sinn machen, sich vorher Gedanken über die beste Klammerung zu machen.

VORGEHEN

Induktiv überlegen wir uns, welche beiden Teilprodukte als letztes miteinander multipliziert werden sollen. Die beiden Operanden dieser finalen Multiplikation hängen davon ab, welche zusammenhängenden Teilstücke der Matrizenfolge schon

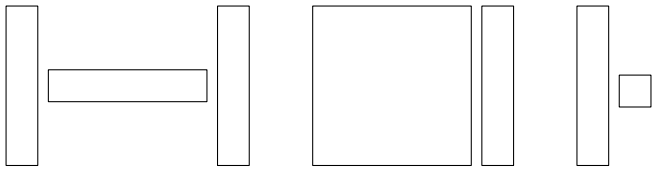


Abb. 3.6 Angenommen, es sind drei Matrizen A_1, A_2 und A_3 gegeben. Dabei haben A_1 und A_3 die Dimensionen $(k \times 1)$, während A_2 die Dimension $(1 \times k)$ hat (links). Klammern wir $((A_1 \times A_2) \times A_3)$, dann berechnen wir zuerst die $(k \times k)$-Matrix $(A_1 \times A_2)$, an die A_3 multipliziert wird (Mitte). Dabei fallen insgesamt $\mathcal{O}(k^2)$ viele Operationen an. Klammern wir dagegen $(A_1 \times (A_2 \times A_3))$, dann berechnen wir zuerst die (1×1)-Matrix $(A_2 \times A_3)$, die an A_1 multipliziert wird (rechts). Dabei fallen lediglich $\mathcal{O}(k)$ viele Operationen an.

miteinander multipliziert wurden. Wenn wir für einen Index i bereits die Matrizen A_1 bis A_i miteinander multipliziert haben und auch schon A_{i+1} bis A_n berechnet haben, dann verbleibt als letztes lediglich $(A_1 \times A_2 \times \cdots \times A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_n)$ zu berechnen. Wir suchen nun das beste i, also die beste Aufteilung in zwei Teilstücke, um den Gesamtaufwand zu minimieren.

Idee für dynamisches Programm

Dazu definieren wir eine zweidimensionale Tabelle M, wobei $M[p,q]$ die Kosten der besten Klammerung von $A_p \times \cdots \times A_q$ angibt. Dieser Eintrag kann durch

$$M[p,q] \leftarrow \min_{p \le i < q} \begin{pmatrix} M[p,i] + M[i+1,q] + \text{Kosten der Multiplikation von} \\ (A_p \times \cdots \times A_i) \text{ mit } (A_{i+1} \times \cdots \times A_q) \end{pmatrix} \quad (67)$$

berechnet werden. Als Randfälle legen wir $M[p,p] \leftarrow 0$ für alle $p \in \{1,\ldots,n\}$ fest.

Berechnungsreihenfolge

In welcher Reihenfolge sollten die Einträge nun berechnet werden? Dazu müssen wir zunächst sicherstellen, dass im Berechnungsschritt keine kreisförmigen Abhängigkeiten entstehen (dies wäre z.B. der Fall, wenn die Berechnung eines Eintrags A die Berechnung von B voraussetzt, die Berechnung von B die Berechnung von C voraussetzt, und die Berechnung von C die Berechnung von A voraussetzt). Um zu sehen, dass der oben beschriebene Berechnungsschritt keine solchen kreisförmigen Abhängigkeiten erzeugt, betrachten wir $q-p$: Die Berechnung von $M[p,q]$ hängt nur von zuvor berechneten Einträgen $M[p',q']$ mit $q'-p' < q-p$ ab. Folglich sollte die Tabelle diagonal gefüllt werden. Wir berechnen also zunächst $M[1,1],\ldots,M[n,n]$, danach $M[1,2],M[2,3],\ldots,M[n-1,n]$, dann $M[1,3],\ldots,M[n-2,n]$ und fahren auf diese Weise fort, bis schlussendlich $M[1,n]$ berechnet wurde. Wie aus dieser Tabelle die entsprechende Reihenfolge ausgelesen werden kann, bleibt dem Leser als Hausaufgabe überlassen.

Laufzeit

Wie schnell ist der soeben beschriebene Algorithmus? Zunächst beobachten wir, dass die DP-Tabelle n^2 viele Einträge besitzt. Zur Berechnung eines Eintrags müssen im schlechtesten Fall bis zu $n-1$ viele andere Einträge betrachtet werden. Somit ergibt sich eine Gesamtlaufzeit $\mathcal{O}(n^3)$.

3.5.1 Exkurs: Multiplikation zweier Matrizen

Problem

Als letztes wollen wir noch kurz überlegen, wie effizient wir zwei $(n \times n)$-Matrizen A und B miteinander multiplizieren können. Das entstehende Produkt $C = A \times B$ hat Grösse $n \times n$, und eine naive Methode zur Multiplikation führt zur Berechnung eines Eintrags in C genau n elementare Multiplikationen aus (genauer: Sind $A = (a_{ij})_{1 \le i,j \le n}$, $B = (b_{ij})_{1 \le i,j \le n}$ und $C = (c_{ij})_{1 \le i,j \le n}$, dann gilt $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$). Also benötigt ein naiver Algorithmus zur Multiplikation zweier $(n \times n)$-Matrizen $\Theta(n^3)$ viele elementare Multiplikationen.

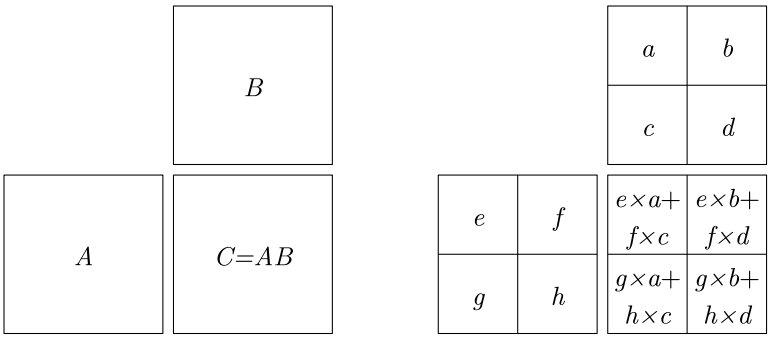


Abb. 3.7 Multiplikation zweier Matrizen A und B und ihre Aufteilung in vier Teilmatrizen.

IDEE

Bereits bei der Multiplikation von Zahlen haben wir gesehen, wie Divide-and-Conquer helfen kann, mit weniger elementaren Operationen auszukommen. Wieder wollen wir die Kosten für die Additionen der einzelnen Teile ignorieren und uns nur auf die Anzahl durchgeführter Multiplikationen konzentrieren. Der Einfachheit halber nehmen wir an, dass $n = 2^k$ für ein $k \in \mathbb{N}_0$ gilt. Wir unterteilen nun A, B und C jeweils in vier gleich grosse $(n/2 \times n/2)$-Teilmatrizen (siehe Abbildung 3.7) und berechnen die Einträge der Teilmatrizen in C rekursiv. Wir berechnen also die acht $(n/2 \times n/2)$-Teilmatrizen $e \times a$, $e \times b$, $f \times c$, $f \times d$, $g \times a$, $g \times b$, $h \times c$ und $h \times d$. Die Anzahl durchgeführter elementarer Multiplikationen bei der Multiplikation zweier $(n \times n)$-Matrizen beträgt damit

$$M(n) = 8M(n/2), M(1) = 1, \tag{68}$$

was die Auflösung $M(n) = 8^{\log_2 n} = n^{\log_2 8} = n^3$ besitzt. Wir haben also bisher noch nichts gewonnen.

VERBESSERUNG

Strassen beobachtete nun, dass es ausreicht, die sieben Produkte $(f-h)\times(c+d)$, $(e-g)\times(a+b)$, $(e+h)\times(a+d)$, $(e+f)\times d$, $(g+h)\times a$, $e\times(b-d)$, $h\times(c-a)$ zu berechnen, denn aus diesen lassen sich alle Teile von C berechnen! So z.B. ergibt sich $e\times b+f\times d$ als Summe aus $(e+f)\times d$ und $e\times(b-d)$. Die anderen drei Teilmatrizen von C können auf ähnliche Weise durch Additionen und Subtraktionen aus den zuvor berechneten Produkten zusammengesetzt werden (nachprüfen!).

LAUFZEIT

Damit verringert sich die Anzahl der elementaren Multiplikationen auf

$$M'(n) = 7M'(n/2), M'(1) = 1, \tag{69}$$

was die Auflösung $M'(n) = 7^{log_2 n} = n^{\log_2 7} \approx n^{2.807}$ besitzt. Das Verfahren von Strassen führt also weniger elementare Multiplikationen als das naive Verfahren aus. Mittlerweile sind Verfahren bekannt, die nur noch etwa $\mathcal{O}(n^{2.373})$ viele Operationen benötigen. Ob man zwei Matrizen auch schneller, vielleicht sogar in Zeit $\Theta(n^2)$ multiplizieren kann, ist unbekannt.

3.6 Subset Sum Problem

PROBLEM-DEFINITION

Alice und Bob sind Geschwister und haben zusammen n Geschenke $\{1, \ldots, n\}$ zu Weihnachten bekommen. Das i-te Geschenk hat dabei den Wert a_i. Da Alice und Bob ähnliche Interessen haben, beschliessen sie die Geschenke gerecht untereinander

aufzuteilen, sodass beide Geschwister Geschenke vom gleichen Wert haben. In diesem Abschnitt wollen wir Algorithmen zur Lösung dieses Problems entwerfen.

FORMALISIERUNG

Dazu nehmen wir an, dass als Eingabe n Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ gegeben sind, und wollen entscheiden, ob eine Auswahl $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,n\} \setminus I} a_i$ existiert. Falls ja, dann soll eine entsprechende Menge I mit dieser Eigenschaft auch ausgegeben werden. Dieses Problem ist unter dem Namen *Subset Sum Problem* bekannt.

SUBSET SUM PROBLEM

NAIVER ALGORITHMUS

Algorithmus 1 (Naiv) Jede Auswahl $I \subseteq \{1, \ldots, n\}$ kann durch einen Bitvektor $b = (b_1, \ldots, b_n) \in \{0, 1\}^n$ der Länge n codiert werden, bei dem $b_i = 1$ genau dann gilt, wenn $i \in I$ ist. Ein naiver Algorithmus würde alle 2^n möglichen Bitvektoren generieren und prüfen, ob die Summe der Werte aller Gegenstände i mit $b_i = 0$ exakt der Summe der Werte aller Gegenstände i mit $b_i = 1$ entspricht. Falls ja, wird die entsprechende Auswahl ausgegeben und der Algorithmus terminiert. Ansonsten wird mit dem nächsten Bitvektor fortgefahren, bis entweder ein entsprechender Bitvektor gefunden wurde, oder aber alle Bitvektoren erfolglos getestet wurden. In diesem Fall existiert keine gerechte Aufteilung aller Geschenke. Da der Algorithmus im schlechtesten Fall 2^n viele Bitvektoren betrachtet und für jeden solchen mindestens einen Schritt ausführen muss, liegt die Laufzeit dieses naiven Algorithmus im schlechtesten Fall zwischen $\Omega(2^n)$ und $\mathcal{O}(n2^n)$.

VERBESSERUNG

Algorithmus 2 (Zerlegung in Teilprobleme) Wir zerlegen die Eingabe zunächst in zwei gleich grosse Teile $a_1, \ldots, a_{n/2}$ sowie $a_{n/2+1}, \ldots, a_n$ (o.B.d.A. sei n gerade). Für jeden dieser zwei Teile generieren wir *alle* erreichbaren $2^{n/2}$ Teilsummen $S_1^k, \ldots, S_{2^{n/2}}^k$ (für $k \in \{1, 2\}$), indem wir wie im naiven Algorithmus über alle Bitvektoren der Länge $n/2$ iterieren, die Werte aller Gegenstände in der entsprechenden Auswahl aufsummieren und die erhaltene Teilsumme abspeichern. Für jeden Teil getrennt sortieren wir dann alle Teilsummen, sodass am Ende $S_1^k \leq S_2^k \leq \cdots \leq S_{2^{n/2}}^k$ für $k \in \{1, 2\}$ gilt. Wir wollen nun prüfen, ob es eine Teilsumme S_i^1 des ersten Teils und eine Teilsumme S_j^2 des zweiten Teils gibt, die zusammen $z := \frac{1}{2} \sum_{i=1}^{n} a_i$ ergeben. Ist $z \notin \mathbb{N}$, dann wissen wir, dass es solche Teilsummen nicht gibt und sind fertig. Ansonsten setzen wir $i \leftarrow 1$ und $j \leftarrow 2^{n/2}$, und unterscheiden drei Fälle:

1) Ist $S_i^1 + S_j^2 = z$, dann haben wir zwei solche Teilsummen gefunden, ermitteln die Menge I aller Gegenstände, deren aufsummierte Werte genau S_i^1 und S_j^2 ergeben, geben I zurück und sind fertig.

2) Ist $S_i^1 + S_j^2 < z$, dann erhöhen wir i um 1.

3) Ansonsten ist $S_i^1 + S_j^2 > z$ und wir verringern j um 1.

Das Verfahren terminiert, wenn entweder $i > n$ oder $j < 1$ gilt; in diesem Fall gibt es keine solchen Teilsummen (und auch keine gerechte Aufteilung der Geschenke).

LAUFZEIT

Die Generierung aller Teilsummen in jedem der zwei Teile benötigt jeweils Zeit $\mathcal{O}(n2^{n/2})$, das Sortieren entsprechend $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$. Ob zwei Teilsummen S_i^1 und S_j^2 mit $S_i^1 + S_j^2 = z$ existieren, kann in Zeit $\mathcal{O}(2^{n/2})$ entschieden werden. Die Gesamtlaufzeit ist also in $\mathcal{O}(n2^{n/2})$, was $\mathcal{O}(n(\sqrt{2})^n)$ entspricht. Dies ist schon bedeutend schneller als der naive Algorithmus, da $\sqrt{2} \approx 1,4142$ gilt.

Man könnte auf die Idee kommen, dass eine Aufteilung in noch kleinere Teilprobleme vielleicht zu einem noch schnelleren Algorithmus führt. Leider ist derzeit

nicht bekannt, ob und wie so etwas konkret funktionieren könnte. Wir brauchen also neue Ideen.

Algorithmus 3 (Dynamische Programmierung) Wie im vorigen Abschnitt definieren wir $z := \frac{1}{2}\sum_{i=1}^{n} a_i$. Wir wollen nun einen Algorithmus entwerfen, der nach dem Prinzip der dynamischen Programmierung arbeitet und feststellt, ob es eine Auswahl I der Gegenstände $\{1,\ldots,n\}$ gibt, sodass $\sum_{i\in I} a_i = z$ gilt, und diese Auswahl ggf. zurückliefert.

Dynamisches Programm

1) *Definition der DP-Tabelle:* Wir benutzen eine zweidimensionale Tabelle T, die in der ersten Dimension von 0 bis n und in der zweiten Dimension von 0 bis z indiziert ist. Jeder Eintrag enthält entweder "wahr" oder "falsch", und $T[k,s]$ gibt an, ob es eine Auswahl I der ersten k Gegenstände $\{1,\ldots,k\}$ gibt, sodass $\sum_{i\in I} a_i = s$ gilt.

2) *Berechnung eines Eintrags:* Wir initialisieren die Tabelle T, indem in $T[0,0]$ der Wert "wahr" und in $T[0,s]$ für alle $s \in \{1,\ldots,z\}$ der Wert "falsch" gespeichert wird (mit $k=0$ Gegenständen kann allein die Teilsumme 0 erreicht werden, aber keine grösseren Teilsummen).

   Zur Berechnung des Eintrags $T[k,s]$ prüfen wir zunächst, ob der k-te Gegenstand überhaupt benutzt werden kann. Ist nämlich $s < a_k$, dann kann dieser Gegenstand nicht dazu benutzt werden, um die Summe s zu erreichen, denn sein Wert ist zu hoch. In diesem Fall gilt also offenbar $T[k,s] = T[k-1,s]$. Ansonsten ist $s \geq a_k$, und wir können den k-ten Gegenstand entweder benutzen (dann muss eine Auswahl der ersten $k-1$ Gegenstände existieren, mit denen die Summe $s-a_k$ erreicht werden kann) oder nicht benutzen (dann muss eine Auswahl der ersten $k-1$ Gegenstände existieren, mit denen die Summe s erreicht werden kann). Wir setzen also

$$T[k,s] \leftarrow \begin{cases} T[k-1,s] & \text{falls } s < a_k, \\ T[k-1,s] \vee T[k-1,s-a_k] & \text{falls } s \geq a_k. \end{cases} \tag{70}$$

   Dabei steht $\vee$ für das logische ODER zweier Wahrheitswerte x und y (hier: $T[k-1,s]$ und $T[k-1,s-a_k]$), das nur dann den Wert "falsch" annimmt, wenn sowohl x als auch y den Wert "falsch" haben.

3) *Berechnungsreihenfolge*: Die Einträge $T[k,s]$ können mit aufsteigendem k und für gleiches k für aufsteigendes s berechnet werden.

4) *Auslesen der Lösung:* Ob eine Auswahl $I \subseteq \{1,\ldots,n\}$ mit $\sum_{i\in I} a_i = z$ existiert, steht am Ende im Eintrag $T[n,z]$. Enthält dieser den Wert "wahr", dann können die Elemente einer solchen Auswahl I wie folgt rekonstruiert werden. Wir setzen zunächst $I \leftarrow \emptyset$. Falls $T[k,s] = T[k-1,s]$ gilt, dann wird der Gegenstand k nicht benutzt, und wir setzen die Rekonstruktion bei $T[k-1,s]$ fort. Ansonsten muss $T[k,s] = T[k-1,s-a_k]$ gelten, folglich fügen wir I den Gegenstand k hinzu und setzen die Rekonstruktion bei $T[k-1,s-a_k]$ fort. Auf diese Weise verfahren wir weiter, bis schlussendlich $k=0$ gilt.

Laufzeit

Der Algorithmus benutzt eine Tabelle der Grösse $(n+1)\cdot(z+1)$, die Berechnung eines jeden Eintrags benötigt nur konstante Zeit, und auch die Lösung kann in Zeit $\mathcal{O}(n)$ ausgelesen werden. Die Gesamtlaufzeit beträgt also $\mathcal{O}(nz)$.

Härte von Problemen Eine Laufzeit von $\mathcal{O}(nz)$ wirkt zunächst polynomiell. Jetzt müssen wir aber gut aufpassen: Anders als sonst ist z eine *Zahl*, und diese kann durch $\lceil \log_2(z+1) \rceil$ viele Bits repräsentiert werden. Die *Eingabelänge* des Problems ist also in $\mathcal{O}(n \log z)$. Ist z selbst polynomiell in n, dann ist auch $n \cdot z$ polynomiell in der Eingabelänge. Ist dagegen z nicht polynomiell in n (z.B., wenn $z = 2^n$ ist), dann ist auch $n \cdot z$ nicht polynomiell in der Eingabelänge. Solch einen Algorithmus nennt man *pseudopolynomiell*. Seine Laufzeit ist genau dann polynomiell, wenn der Wert *aller* in der Eingabe vorkommenden Zahlen polynomiell in der Eingabelänge beschränkt ist.

EINGABELÄNGE

PSEUDO-POLYNOMIELL

Es ist nicht bekannt, ob für das Subset Sum Problem ein Algorithmus mit echt polynomieller Laufzeit existiert. Um dies näher zu erläutern, müssen wir kurz ausholen und zwei Klassen von Problemen definieren. Die Klasse P ist als Menge aller in Polynomialzeit lösbarer Probleme definiert. Für jedes Problem in P gibt es also einen Algorithmus, der in polynomieller Zeit (in der Länge der Eingabe) eine Lösung für das Problem berechnet. Die Klasse NP ist als Menge aller *nichtdeterministisch polynomiell* lösbarer Probleme definiert.[1] Das sind alle Probleme, für die man eine Lösung in polynomieller Zeit *verifizieren* kann. Angenommen, wir betrachten das Subset Sum Problem für die Eingabe $n = 3, a_1 = 8, a_2 = 9, a_3 = 17$. Hätten wir ein gutartiges *Orakel*, das auf irgendeine Art die Lösung $\{1, 2\}$ ermittelt und uns mitteilt, dann können wir in Polynomialzeit prüfen, dass diese wirklich eine gleichmässige Aufteilung der Werte $\{a_1, \ldots, a_n\}$ darstellt (indem wir die Werte der Auswahl der vorgegebenen Lösung aufsummieren und prüfen, ob diese Summe exakt halb so gross wie die Summe aller Werte $a_1, \ldots, a_n$ ist). Natürlich ist P eine Teilmenge von NP, denn können wir eine Lösung selbst in Polynomialzeit ermitteln, dann können wir für eine gegebene Lösung auch in Polynomialzeit prüfen, ob sie das Problem löst oder nicht. Die Frage, ob P = NP gilt, ist die bedeutendste unlöste Frage in der theoretischen Informatik. Oftmals wird vermutet, dass die Gleichheit nicht gilt. Für das Subset Sum Problem ist bekannt, dass es in NP liegt. Es ist aber nicht bekannt, ob es auch in P liegt. Unter der Annahme P ≠ NP gibt es also für dieses Problem keinen Algorithmus mit einer echt polynomiellen Laufzeit; auf viel mehr als einen pseudopolynomiellen Algorithmus können wir im Moment nicht hoffen.

NICHTDETERMINISTISCH POLYNOMIELL

ORAKEL

3.7 Rucksackproblem

PROBLEMDEFINITION

Ein Tierarzt fährt zu seinen Patienten und hat eine Menge von n Gegenständen $\{1, \ldots, n\}$ zur Verfügung, die bei einer Behandlung nützlich sein könnten. Jeder Gegenstand i hat den Nutzwert (engl. *value*) v_i und das Gewicht (engl. *weight*) w_i. Leider erlaubt das benutzte Fahrzeug nur eine Zuladung des Gewichts W.[2] Daher wollen wir eine Auswahl $I \subseteq \{1, \ldots, n\}$ der Gegenstände berechnen, deren Summe der Gewichte $\sum_{i \in I} w_i$ höchstens W beträgt, und deren gesamter Nutzen $\sum_{i \in I} v_i$ maximal ist. Dieses Problem ist unter dem Namen *Rucksackproblem* bekannt.

Eine gierige Heuristik Eine naheliegende Idee zur Lösung besteht darin, die Gegenstande zunächst absteigend nach Nutzen pro Gewichtseinheit v_i/w_i zu sortieren und die Gegenstände in der entsprechenden Reihenfolge zu betrachten. Kann der

[1] Der Buchstabe "N" in NP steht für "nichtdeterministisch", und **nicht** für das Wort "nicht"!

[2] Wir können o.B.d.A. annehmen, dass der Fahrer Gewicht 0 hat, indem wir sein Gewicht von der maximal erlaubten Zuladung subtrahieren.

aktuell betrachtete Gegenstand i zur aktuellen Auswahl I (initial, $I = \emptyset$) hinzugefügt werden, ohne dass das Gesamtgewicht von W überschritten wird, dann setzen wir $I \leftarrow I \cup \{i\}$. Würde das Hinzufügen des aktuellen Gegenstands i dagegen dazu führen, dass das maximal zulässige Gesamtgewicht W überschritten wird, dann wird der Gegenstand einfach nicht hinzugefügt.

Eine solche Strategie wird "gierig" (engl. *greedy strategy*) genannt, da sie stets die Entscheidung trifft, die im Moment die beste zu sein scheint. Der Algorithmus ist offenbar sehr schnell, da für das Sortieren Zeit $\Theta(n \log n)$ und für die anschliessende Berechnung der Auswahl lediglich Zeit $\Theta(n)$ gebraucht werden.

LAUFZEIT

Leider kann dieser Algorithmus Lösungen berechnen, die beliebig viel schlechter als die optimale Lösung sind. Dazu betrachten wir zwei Gegenstände mit den Nutzwerten $v_1 = 1$ und $v_2 = W - 1$ und den Gewichten $w_1 = 1$ und $w_2 = W$, und setzen das maximal zulässige Gewicht auf W. Eine optimale Auswahl I_{opt} besteht offenbar nur aus dem zweiten Gegenstand (mit einem Nutzen von $W - 1$), die obige gierige Heuristik würde aber stets die Auswahl zurückliefern, die nur den ersten Gegenstand enthält (denn $v_1/w_1 = 1 > 1 - \frac{1}{W} = v_2/w_2$). Diese Auswahl hat lediglich den Nutzen 1, was $W - 1$ Mal schlechter als der Wert der besten Auswahl ist.

QUALITÄT DER LÖSUNG

Eine erste DP-Lösung Analog zum zuvor besprochenen Subset Sum Problem wollen wir nun dynamische Programmierung zur Lösung des Rucksackproblems benutzen. Dazu nehmen wir an, dass wir uns eine kleinere Gewichtsgrenze vorgeben und dann schauen, ob wir mit den ersten i Gegenständen einen Mindestwert erreichen können. Formaler benutzen wir eine dreidimensionale DP-Tabelle machbar(i, w, v), die Wahrheitswerte (also entweder "wahr" oder "falsch") enthält, und wir setzen machbar(i, w, v) ist genau dann "wahr", wenn es eine Auswahl (also eine Teilmenge) der ersten i Gegenstände gibt, deren Gesamtgewicht höchstens w und deren Nutzen mindestens v ist. Initial setzen wir machbar$[i, w, 0] \leftarrow$ "wahr" für alle $i \geq 0$ und alle $w \geq 0$ und machbar$[0, w, v] \leftarrow$ "false" für alle $w \geq 0$ und alle $v > 0$. Einen Eintrag der Tabelle berechnen wir nun durch

DP-TABELLE

BERECHNUNG EINES EINTRAGS

$$\text{machbar}[i, w, v] \leftarrow \text{machbar}[i-1, w, v] \vee \text{machbar}[i-1, w-w_i, v-v_i], \qquad (71)$$

falls $w \geq w_i$ und $v \geq v_i$ gilt, und ansonsten entsprechend durch

$$\text{machbar}[i, w, v] \leftarrow \text{machbar}[i-1, w, v]. \qquad (72)$$

Wir berechnen die Einträge nach aufsteigendem i (für $i = 0, \ldots, n$), Einträge mit gleichem i nach aufsteigendem w (für $w = 0, \ldots, W$) und Einträge mit gleichem i und w nach aufsteigendem v (für $v = 0, \ldots, \sum_{i=1}^{n} v_i$). Der maximal erreichbare Wert einer Auswahl ist der grösste Index v, für den Indizes i und w mit machbar$[i, w, v] =$ "true" existieren. Die Berechnung der entsprechenden Gegenstände einer besten Auswahl erfolgt analog zum dynamischen Programm für das Subset Sum Problem. Insgesamt hat unser dynamisches Programm eine Laufzeit von $\mathcal{O}(n \cdot W \cdot \sum_{i=1}^{n} v_i)$.

BERECHNUNGSREIHENFOLGE

AUSLESEN DER LÖSUNG

Verbesserung Betrachten wir die soeben definierte DP-Tabelle genauer, dann stellen wir Folgendes fest:

- Ist machbar$(i, v, w) =$ "true", dann ist auch machbar$(i', v', w') =$ "true" für alle $i' \geq i$, alle $v' \leq v$ und alle $w' \geq w$, und analog

- Ist machbar(i, v, w) = "false", dann ist auch machbar(i', v', w') = "false" für alle $i' \leq i$, alle $v' \geq v$ und alle $w' \leq w$.

DP-Tabelle

Folglich ist braucht die DP-Tabelle keine drei Dimensionen. Stattdessen könnten wir auch eine zweidimensionale Tabelle maxWert benutzen, die in der ersten Dimension von 0 bis n und in der zweiten Dimension von 0 bis W indiziert ist, und in der ein Eintrag maxWert$[i, w]$ den maximal erreichbaren Nutzen angibt, wenn nur aus den Gegenständen $\{1, \ldots, i\}$ ausgewählt werden darf, und das Gewicht der Auswahl maximal w betragen darf. Zur Initialisierung setzen wir maxWert$[0, w] \leftarrow 0$ (wird kein Gegenstand benutzt, dann ist der maximal erreichbare Nutzen 0). Für $i > 0$ berechnen wir dann

Berechnung eines Eintrags

$$\text{maxWert}[i, w] \leftarrow \max\Big\{\text{maxWert}[i-1, w], \text{maxWert}[i-1, w-w_i] + v_i\Big\}, \quad (73)$$

falls $w \geq w_i$ ist (wie im dynamischen Programm für das Subset Sum Problem kann der i-te Gegenstand entweder genommen werden, oder nicht), und ansonsten wird

$$\text{maxWert}[i, w] \leftarrow \text{maxWert}[i-1, w] \quad (74)$$

Auslesen der Lösung

gesetzt. Der maximal erreichbare Nutzen einer Auswahl ist am Ende im Eintrag maxWert$[n, W]$ gespeichert, und von diesem Eintrag aus können die Gegenstände einer Auswahl mit diesem Nutzen genau wie im dynamischen Programm für das Subset Sum Problem rekonstruiert werden.

Laufzeit

Die Laufzeit dieses dynamischen Programms ist in $\Theta(nW)$.

Eine Polynomialzeitapproximation für das Rucksackproblem Für das Rucksackproblem haben wir im Wesentlichen drei Algorithmen kennengelernt: Eine Greedy-Heuristik, die sehr schnell ist, aber sehr schlechte Ergebnisse liefern kann, sowie zwei dynamische Programme, die eine exakte Lösung bestimmen, dafür aber pseudopolynomielle Zeit benötigen.

Im Folgenden wollen wir versuchen, eine Näherungslösung zu berechnen. Wir geben uns damit zufrieden, eine sehr gute, ggf. aber nicht optimale, Lösung zu finden. Dafür möchten wir aber auch eine Garantie haben, dass die berechnete Lösung nicht wesentlich schlechter als die optimale Lösung ist. Ist eine Zahl $\varepsilon \in (0, 1)$ gegeben, dann suchen wir eine Auswahl I mit

$$\sum_{i \in I} v_i \geq (1 - \varepsilon) \sum_{i \in OPT} v_i, \quad (75)$$

wobei OPT eine optimale Auswahl für die entsprechende Eingabe bezeichnet. Wir möchten also, dass der Nutzen unserer Lösung garantiert nur um einen *vorgegebenen* Faktor ε von der optimalen Lösung abweicht. Beim Gewicht der Auswahl dürfen natürlich keine Kompromisse gemacht werden: Die Summe der Gewichte aller Gegenstände in I darf W niemals übersteigen.

Zum Entwurf eines solchen *Approximationsalgorithmus* überlegen wir zunächst, dass wir das soeben vorgestellte dynamische Programm auch anders hätten formulieren können. Statt für eine gegebene Gewichtsschranke w den maximal erreichbaren Nutzen zu ermitteln, könnten wir für einen gegebenen Nutzen v auch fragen, wie viel Gewicht eine Auswahl mit diesem Nutzen mindestens haben muss. Dieses alternative dynamische Programm benutzt also wieder eine zweidimensionale Tabelle, diesmal minGew genannt, und ein Eintrag minGew$[i, v]$ gibt das minimale Gewicht

DP-Tabelle

einer Auswahl an, die eine Teilmenge der ersten i Gegenstände ist und einen Nutzen von exakt v hat. Dabei sind $0 \leq i \leq n$ und $0 \leq v \leq \sum_{i=1}^{n} v_i$ (diese Summe ist der theoretisch maximal erreichbare Nutzen, wenn alle Gegenstände in der Auswahl enthalten wären). Aufgrund der geänderten Bedeutung eines Eintrags brauchen wir natürlich auch eine andere Initialisierung. Wir setzen also $\text{minGew}[0,0] \leftarrow 0$ (um den Nutzen 0 unter Verwendung keinerlei Gegenstände zu erzielen, braucht es Gewicht 0), und $\text{minGew}[0,v] \leftarrow \infty$ für alle $v > 0$ (da mit keinen Gegenständen kein strikt positiver Nutzen erzielt werden kann, wird das Gewicht einer potentiellen Auswahl auf ∞ gesetzt). Einen Eintrag $\text{minGew}[i,v]$ berechnen wir mit

Berechnung eines Eintrags

$$\text{minGew}[i,v] \leftarrow \min\Big\{\text{minGew}[i-1,v], \text{minGew}[i-1,v-v_i] + w_i\Big\}, \tag{76}$$

falls $v \geq v_i$ ist, und ansonsten setzen wir

$$\text{minGew}[i,v] \leftarrow \text{minGew}[i-1,v]. \tag{77}$$

Der maximal erreichbare Nutzen ist der grösste Index v mit $\text{minGew}[n,v] \leq W$. Das Auslesen der Lösung erfolgt dann analog zum direkt zuvor diskutierten dynamischen Programm. Die Laufzeit dieses dynamischen Programms ist in $\Theta(n \sum_{i=1}^{n} v_i)$, ist also nach wie vor pseudopolynomiell.

Auslesen der Lösung

Laufzeit

Eine pseudopolynomielle Laufzeit wird polynomiell, wenn alle vorkommenden Werte durch ein Polynom in der Eingabelänge beschränkt werden können. Können wir die Nutzwerte v_i also so verkleinern, sodass die Laufzeit polynomiell wird, und die Werte trotzdem noch hinreichend genau sind, um eine gute Lösung zu finden? Dazu nehmen wir an, es gäbe eine *geeignet gewählte* Zahl K und ersetzen die Nutzwerte v_i durch die gerundeten Nutzwerte $\tilde{v}_i := \lfloor v_i/K \rfloor$. Wäre also etwa $K = 1000$, dann würden von jedem Nutzwert die letzten drei Ziffern in einer Dezimaldarstellung abgeschnitten. Wie K "geeignet gewählt" werden kann, wird in Kürze erläutert. Wir lassen dann das soeben beschriebene dynamische Programm auf einer geänderten Eingabe (im Folgenden Eingabe$\sim$) mit n Gegenständen laufen, wobei der i-te Gegenstand wie zuvor das Gewicht w_i, aber einen geänderten Nutzen $\tilde{v}_i$ hat. Als Gewichtsschranke benutzen wir wieder W. Nun beobachten wir:

1) Jede Auswahl von Gegenständen in Eingabe$\sim$ ist auch in der ursprünglichen Eingabe gültig, denn jeder Gegenstand in Eingabe$\sim$ hat genau das gleiche Gewicht wie in der ursprünglichen Eingabe, und das Maximalgewicht der Auswahl beträgt noch immer W.

2) Die Laufzeit des Algorithmus ist durch $\mathcal{O}(n^2 v_{max}/K)$ nach oben beschränkt (wobei $v_{max} = \max\{v_i \mid i \in \{1,\ldots,n\}\}$ der maximale Nutzen eines Gegenstands in der ursprünglichen Eingabe ist), denn $\sum_{i=1}^{n} \tilde{v}_i = \sum_{i=1}^{n} \lfloor v_i/K \rfloor \leq \sum_{i=1}^{n} v_i/K = \frac{1}{K} \sum_{i=1}^{n} v_i \leq \frac{n}{K} v_{max}$.

Wie schlecht kann die Lösung dieser Approximation werden? Beim Abrunden der Nutzwerte verlieren wir höchstens K, d.h. es gilt

$$v_i - K \leq K \cdot \lfloor \frac{v_i}{K} \rfloor = K \cdot \tilde{v}_i \leq v_i. \tag{78}$$

Wir bezeichnen nun mit OPT ist eine exakte optimale Lösung für die ursprüngliche

Eingabe, und mit $OPT\sim$ eine exakte optimale Lösung für Eingabe$\sim$. Dann gilt

$$\Big(\sum_{i\in OPT} v_i\Big) - n\cdot K \overset{(*)}{\leq} \sum_{i\in OPT} v_i - \sum_{i\in OPT} K \leq \sum_{i\in OPT}(v_i - K) \tag{79}$$

$$\leq \sum_{i\in OPT} K\cdot\tilde{v}_i = K\cdot\sum_{i\in OPT}\tilde{v}_i \overset{(**)}{\leq} K\cdot\sum_{i\in OPT\sim}\tilde{v}_i \tag{80}$$

$$= \sum_{i\in OPT\sim} K\cdot\tilde{v}_i \leq \sum_{i\in OPT\sim} v_i. \tag{81}$$

Die Ungleichung $(*)$ gilt aufgrund der Tatsache, dass OPT höchstens n Gegenstände enthalten kann, folglich also $|OPT|\leq n$ gilt. Die Ungleichung $(**)$ gilt, da $OPT\sim$ optimal bezüglich Eingabe$\sim$ ist, folglich muss die Summe der Nutzwerte aller Gegenstände in $OPT\sim$ mindestens so gross wie die Summe der Nutzwerte aller Gegenstände in OPT sein.

Wir hatten früher gefordert, dass der Gesamtnutzen der von unserem Approximationsalgorithmus berechneten Auswahl höchstens um einen Faktor ε vom Optimum abweicht. Dies kann erreicht werden, indem K so gewählt wird, dass $nK = \varepsilon\sum_{i\in OPT} v_i$ gilt. Eine optimale Auswahl für die ursprünglichen Nutzwerte kennen wir natürlich nicht. Wir wissen aber bereits, dass der maximal erreichbare Nutzen mindestens v_{max} ist, denn unter der sinnvollen Annahme, dass jeder Gegenstand ein Gewicht hat, das die Gewichtsschranke nicht übersteigt (sonst kann er einfach aus der Eingabe verworfen werden), können wir den Gegenstand mit Nutzen v_{max} immer wählen und haben eine Auswahl mit diesem Wert. Also ist $\sum_{i\in OPT} v_i \geq v_{max}$, und wir wählen K nun so, dass $nK = \varepsilon\cdot v_{max}$ gilt, also

$$K = \frac{\varepsilon\cdot v_{max}}{n}. \tag{82}$$

Da K etwas kleiner als der ursprüngliche Wert $(\varepsilon\sum_{i\in OPT} v_i)/n$ ist, wird die Approximation etwas besser, und die Laufzeit geringfügig schlechter. Wir haben also einen Algorithmus gefunden, der eine $(1-\varepsilon)$-Approximation für eine Eingabe des Rucksackproblems in Zeit $\mathcal{O}(n^2\cdot\frac{v_{max}}{K}) = \mathcal{O}(\frac{n^3}{\varepsilon})$ berechnet. Die Laufzeit hängt also lediglich noch von der Anzahl der gegebenen Gegenstände und der Approximationsgüte ε ab und ist völlig unabhängig von den konkreten Gewichten und Nutzwerten.

LAUFZEIT

APPROXIMATIONSSCHEMA

Solch einen Algorithmus nennt man ein *Approximationsschema*: Sobald wir ein ε gewählt haben, ist ε quasi eine Konstante und der Algorithmus hat dann polynomielle Laufzeit. Die Laufzeit ist also ein Polynom in n und in $\frac{1}{\varepsilon}$. In diesem Fall nennt man das Approximationsschema dann ein *voll-polynomielles Approximationsschema* (engl. *FPTAS – fully polynomial-time approximation scheme*).

Man kann zeigen, dass das Rucksackproblem wie das Subset Sum Problem nicht in Polynomialzeit lösbar ist, wenn $\mathsf{P}\neq\mathsf{NP}$ gilt. Die Existenz eines FPTAS ist also das Beste, was wir für ein solches Problem erwarten können, denn es erlaubt die Berechnung einer beliebig guten Approximation in Polynomialzeit. Viele andere Probleme haben kein FPTAS. Manchmal haben sie aber Approximationsalgorithmen, bei deren Laufzeit der Faktor $\frac{1}{\varepsilon}$ nicht als multiplikativer Faktor, sondern als Exponent auftaucht, z.B. wie in $\mathcal{O}(n^{\frac{1}{\varepsilon}})$. Für ein fixes ε ist dies dann zwar immer noch polynomiell, aber die Potenz hängt nun von ε ab.

Kapitel 4

Datenstrukturen für Wörterbücher

4.1 Abstrakte Datentypen

Der bisherige Fokus der Vorlesung lag besonders auf dem Entwurf effizienter Algorithmen für gegebene Problemstellungen (z.B. die Sortierung eines Arrays). Wir wollen heute Datenstrukturen und Algorithmen für *abstrakte Datentypen* (ADT) entwerfen. Abstrakte Datentypen dienen der Verwaltung einer Menge von *Objekten* (z.B. Zahlen, Zeichenketten, Datensätzen, usw.). Dazu stellt jeder ADT einige *Operationen* zur Verfügung, die die Menge der verwalteten Objekte verändern oder Objekte mit gewissen Eigenschaften zurückliefern können. Ein ADT beschreibt also nur die grundsätzliche Funktionalität, aber keine konkrete Implementierung. Für eine Implementierung eines ADT werden Datenstrukturen und Algorithmen gemeinsam entworfen: Während Datenstrukturen beschreiben, wie die Daten systematisch gespeichert werden können, beschreiben Algorithmen die Implementierung der einzelnen Operationen. Wir werden im Folgenden einige abstrakte Datentypen kennenlernen und erläutern, wie sie durch geeignete Datenstrukturen und Algorithmen implementiert werden können.

Abstrakter Datentyp

Stapel (Stack) Ein *Stack* ist ein ADT, der die folgenden Operationen unterstützt:

Operationen

- Push(x, S): Legt das Objekt x oben auf den Stack S.
- Pop(S): Entfernt das oberste Objekt vom Stack S und liefert es zurück. Enthält S keine Objekte, soll **null** zurückgeliefert werden.
- Top(S): Liefert das oberste Objekt vom Stack S zurück, entfernt es aber nicht aus S. Enthält S keine Objekte, soll **null** zurückgeliefert werden.
- isEmpty(S): Liefert genau dann "true" zurück, wenn der Stack S leer ist, d.h. keine Objekte mehr enthält, und "false" ansonsten.
- EmptyStack: Liefert einen leeren Stack zurück.

Man beachte, dass sowohl Pop(S) als auch Top(S) auch dann noch funktionieren, wenn der Stack leer ist (sie liefern dann einfach **null** zurück). In einer Implementierung in einer echten Programmiersprache müsste man entsprechend überlegen, wie mit einer derartigen Situation umgegangen werden sollte.

Einen Stack kann man sich anschaulich wie einen Stapel von (Porzellan)tellern vorstellen: Zum Essen nimmt man den obersten Teller weg und legt ihn nach dem Spülen wieder ganz oben auf den Stapel zurück.

Anschauliche Vorstellung

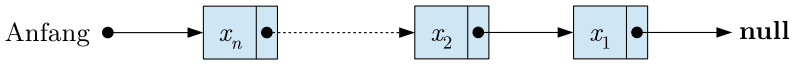


Abb. 4.1 Ein durch eine lineare Liste implementierter Stack, auf dem die Operationen EmptyStack, Push(x_1), Push(x_2), ..., Push(x_n) in genau dieser Reihenfolge ausgeführt wurden.

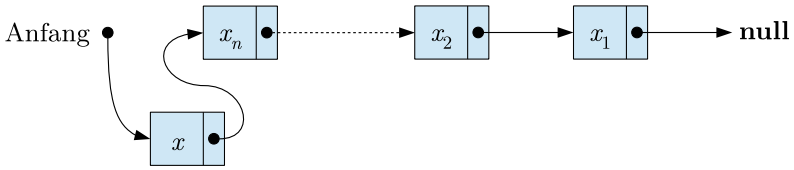


Abb. 4.2 Hinzufügen des Objekts x zu einem Stack, der die Objekte $x_1, \ldots, x_n$ verwaltet.

Datenstruktur

Implementieren kann man einen Stack durch eine einfach verkettete Liste, wobei jedes Listenelement genau ein Objekt x speichert, und zwar in umgekehrter Reihenfolge, in der die Objekte mittels Push hinzugefügt wurden (siehe Abbildung 4.1). Wir verwalten also einen Anfangszeiger $\vec{p_A}$, der auf das Listenelement zeigt, das das zuletzt hinzugefügte Objekt speichert, und jedes Listenelement zeigt auf das Listenelement, das das zeitlich direkt vorher hinzugefügte Objekt enthält. Das Listenelement, das das am frühesten hinzugefügte Objekt speichert, hat einen Nullzeiger, um das Ende der Liste anzuzeigen. Die Operationen des Stacks können nun wie folgt implementiert werden:

Implementierung der Operationen

- Push(x, S): Erzeuge ein neues Listenelement, das x enthält, und das auf das Listenelement zeigt, auf das momentan $\vec{p_A}$ zeigt. Modifiziere den Anfangszeiger $\vec{p_A}$ so, dass er auf das soeben erzeugte Listenelement zeigt (siehe Abbildung 4.2).
- Pop(S): Ist $l =$ **null**, dann gib **null** zurück. Merke ansonsten das Objekt x, das in dem Listenelement l gespeichert ist auf das $\vec{p_A}$ derzeit zeigt. Sei l' das nachfolgende Listenelement von l (auf das l zeigt, l' kann also auch **null** sein). Modifiziere den Anfangszeiger $\vec{p_A}$ so, dass er auf l' zeigt, entferne l aus dem Speicher und gib x zurück.
- Top(S): Ist $l =$ **null**, dann gib **null** zurück. Gib ansonsten das Objekt x, das in dem Listenelement gespeichert ist auf das $\vec{p_A}$ derzeit zeigt, zurück.
- isEmpty(S): Liefere "true" genau dann zurück, wenn $\vec{p_A} =$ **null** ist.
- EmptyStack: Setze $\vec{p_A} \leftarrow$ **null** und gib $\vec{p_A}$ zurück.

Laufzeit

Jede dieser Operationen ist in Zeit $\mathcal{O}(1)$ durchführbar. Wenn im Vorfeld bereits die maximale Anzahl zu verwaltender Objekte bekannt ist, kann man einen Stack auch leicht unter Verwendung eines Arrays anstelle einer einfach verketteten Liste implementieren, was die asymptotische Laufzeit der Operationen nicht ändert. Die Details auszuarbeiten bleibt als Übung dem Leser überlassen.

Implementierung mit Arrays

Operationen

Schlange (Queue) Eine *Queue* ist ein ADT, der die folgenden Operationen unterstützt:

- Enqueue(x, Q): Fügt das Objekt x hinten an die Queue Q an.

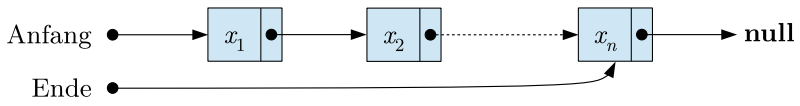


Abb. 4.3 Eine durch eine lineare Liste implementierter Queue, auf der die Operationen EmptyQueue, Enqueue(x_1), Enqueue(x_2), ..., Enqueue(x_n) in genau dieser Reihenfolge ausgeführt wurden.

- Dequeue(Q): Entfernt das am weitesten vorn stehende Objekt aus der Queue Q und liefert es zurück. Enthält Q keine Objekte, soll **null** zurückgeliefert werden.
- Front(Q): Liefert das am weitesten vorn stehende Objekt aus der Queue Q zurück, entfernt es aber nicht aus Q. Enthält Q keine Objekte, soll **null** zurückgeliefert werden.
- isEmpty(Q): Liefert genau dann "true" zurück, wenn die Queue Q leer ist, d.h. keine Objekte mehr enthält, und "false" ansonsten.
- EmptyQueue: Liefert eine leere Queue zurück.

Vergleich mit Stacks

Der Unterschied zu einem Stack ist also, dass Stacks Objekte immer vorn anfügen und auch vorn wieder wegnehmen, während Queues Objekte hinten anfügen und vorn wegnehmen. Während Pop(S) also das zuletzt hinzugefügte Objekt zurückliefert, liefert Dequeue(Q) das am längsten in der Queue gespeicherte Objekt zurück.

Anschauliche Vorstellung

Eine Queue kann man sich anschaulich wie eine (idealisierte) Schlange im Supermarkt vorstellen, in der sich niemand vordrängelt.

Datenstruktur

Queues können analog zu Stacks durch einfach verkettete Listen implementiert werden (siehe Abbildung 4.3). Anders als bei Stacks werden die Objekte aber in genau der Reihenfolge gespeichert, in der sie eingefügt wurden (und nicht in umgekehrter Reihenfolge). Zudem verwalten wir nicht nur einen Zeiger $\vec{p_A}$ auf das erste Listenelement, sondern auch noch einen Zeiger $\vec{p_E}$ auf das letzte Listenelement.

Implementierung

Auf diese Art kann Enqueue(x, Q) in Zeit $\mathcal{O}(1)$ implementiert werden, indem zunächst ein neues Listenelement l erzeugt wird, das x speichert. Ist $\vec{p_E} \neq$ **null**, dann folgt man $\vec{p_E}$, legt l als Nachfolger dieses Listenelements fest und lässt $\vec{p_E}$ auf l zeigen. Waren vorher $\vec{p_A} = \vec{p_E} =$ **null**, dann lässt man diese direkt auf das neue Listenelement l zeigen lassen. Dequeue(Q), Front(Q), isEmpty(Q) sowie EmptyQueue können analog zu den entsprechenden Operationen für Stacks implementiert werden, man muss nur $\vec{p_E}$ ggf. anpassen (der Zeiger muss u.U. auf **null** gesetzt werden).

Laufzeit

Arbeitet man alle Implementierungsdetails gründlich aus, wird man feststellen, dass wie bei Stacks alle Operationen in Zeit $\mathcal{O}(1)$ durchführbar sind. Ist die maximale Anzahl zu verwaltender Objekte im Voraus bekannt, können Queues auch durch Arrays implementiert werden.

Prioritätswarteschlange (Priority Queue) Eine *Priority Queue* funktioniert ähnlich wie eine Queue, wir können Objekte nun aber mit einer Priorität einfügen. Enqueue(x, Q) wird dann von einer Operation Insert(x, p, Q) ersetzt, wobei x das einzufügende Objekt, $p \in \mathbb{N}$ die Priorität von x und Q die Priority Queue darstellen. Analog wird Dequeue(Q) durch eine Operation Extract-Max(Q) ersetzt, die das Objekt mit der höchsten Priorität aus der Priority Queue Q entfernt und zurückliefert. Der Einfachheit halber verbieten wir, dass zwei verschiedene Objekte

Operationen

x und x' die gleiche Priorität haben, sodass die Ausgabe von EXTRACT-MAX(Q) immer wohldefiniert ist.

IMPLEMENTIERUNG

Schon früher haben wir eine Implementierung für Priority Queues kennengelernt, nämlich Heaps (siehe Abschnitt 2.4). Wir können die Prioritäten als Schlüssel benutzen und die Objekte selbst in einem zweiten Array speichern (dessen Elemente beim Bewegen von Schlüsseln entsprechend mitbewegt werden müssen). Wie man bei einem Heap das Maximum extrahiert, haben wir bereits früher gesehen. Man überlege sich als Hausaufgabe, wie die Operation INSERT(x, p, Q) durch Heaps implementiert werden kann. Ebenso überlege man sich, dass sowohl EXTRACT-MAX(Q) als auch INSERT(x, p, Q) nur Zeit $\mathcal{O}(\log n)$ benötigen, wenn die Priority Queue n Objekte verwaltet.

OPERATIONEN

Multistapel (Multistack) Ein *Multistack* unterstützt neben allen Operationen, die ein herkömmlicher Stack unterstützt, noch eine weitere:

- MULTIPOP(k, S): Entfernt die k zuletzt hinzugefügten Objekte aus S und liefert sie (nach Zeitpunkt der Hinzufügung absteigend sortiert) zurück. Enthält S weniger als k Elemente, werden entsprechend weniger Elemente entfernt und zurückgeliefert. Insbesondere wird **null** zurückgeliefert wenn S vor Aufruf der Operation keine Elemente enthielt.

DATENSTRUKTUR IMPLEMENTIERUNG

Ein Multistack kann genau wie herkömmliche Stacks durch einfach verkettete Listen implementiert werden; die Implementierung von MULTIPOP(k, S) erfolgt analog zur Implementierung von POP(S).

LAUFZEIT

Einzig die Laufzeit von MULTIPOP(k, S) beträgt jetzt $\mathcal{O}(k)$ statt $\mathcal{O}(1)$ wie die von POP(S), denn es werden ja k Objekte entfernt und zurückgegeben.

Die obere Schranke von $\mathcal{O}(k)$ ist auch tatsächlich exakt, denn wenn S mindestens k Elemente enthält, wird wirklich Zeit $\Theta(k)$ benötigt, um alle diese Elemente von S zu entfernen. Das heisst: Führen wir auf einem Stack S, der n Objekte verwaltet, n Mal MULTIPOP(k, S) aus, dann kostet dies Zeit $\mathcal{O}(n^2)$ (denn jeder Aufruf von MULTIPOP(k, S) kann Zeit $\mathcal{O}(n)$ brauchen, wenn $k \in \mathcal{O}(n)$ ist). Andererseits können wir keine Folge von Multistack-Operationen angeben, die tatsächlich quadratische Kosten hat, denn jeder teure (langwierige) Aufruf von MULTIPOP entfernt gleichzeitig viele Elemente vom Stack. Zukünftige Aufrufe von MULTIPOP sind daher möglicherweise nicht mehr ganz so teuer (langwierig).

BESSERE ANALYSE (ANSCHAULICH)

Für eine genauere Analyse nehmen wir vereinfachend an, dass sowohl PUSH(x, S) als auch POP(S) Kosten 1 haben (der Liste muss genau ein Element hinzugefügt oder entfernt werden), während MULTIPOP(k, S) Kosten k hat (der Liste müssen k Elemente entfernt werden). Gedanklich können wir uns nun vorstellen, dass wir bei jedem Aufruf von PUSH(x, S) einen Schweizer Franken auf ein Bankkonto legen. Wird ein Objekt irgendwann von S entfernt (entweder durch POP oder durch MULTIPOP), dann zahlen wir die Entfernung dieses Objekts vom Bankkonto. Da die Kosten zur Entfernung pro Objekt sowohl in POP als auch in MULTIPOP exakt 1 sind, wir pro Objekt genau einen Schweizer Franken auf das Bankkonto eingezahlt haben und ein Objekt nur dann entfernen, wenn es zuvor irgendwann eingefügt wurde, wird der Kontostand auch niemals negativ. Mit dieser alternativen Sichtweise hat PUSH also Kosten 2, während sowohl POP als auch MULTIPOP Kosten 0 haben. Folglich sind die durchschnittlichen Kosten für jede dieser drei Operationen noch immer in $\mathcal{O}(1)$.

AMORTISIERTE ANALYSE

Diese Analysetechnik wollen wir nun noch ein wenig formalisieren. Dazu nehmen wir an, wir haben eine Datenstruktur, auf der n Operationen $O_1, \ldots, O_n$ in dieser Reihenfolge ausgeführt werden (z.B. die $n = 7$ Operationen $O_1 =$ EMPTYSTACK,

$O_2 = \textsc{Push}(15, S)$, $O_3 = \textsc{Push}(7, S)$, $O_4 = \textsc{Push}(200, S)$, $O_5 = \textsc{Pop}(S)$, $O_6 = \textsc{Push}(2, S)$ sowie $O_7 = \textsc{Multipop}(3, S)$). Manche dieser Operationen (im vorigen Beispiel alle) verändern die Datenstruktur. Die i-te Operation habe reale Kosten t_i (im vorigen Beispiel haben alle Operationen i reale Kosten $t_i = 1$, mit Ausnahme der letzten Operation, die reale Kosten $t_7 = 3$ hat). Wir modellieren den Stand unseres Bankkontos durch eine *Potentialfunktion* Φ_i, die den Kontostand *nach* Durchführung der ersten i Operationen angibt. Da wir annehmen, dass das Bankkonto zu Beginn leer ist, setzen wir $\Phi_0 := 0$. Die Potentialfunktion muss so gewählt werden, dass stets $\Phi_i \geq 0$ gilt. Anders als im echten Leben darf das Bankkonto also *nicht* überzogen werden! Nun definieren wir die *amortisierten Kosten* der i-ten Operation als

Amortisierte Kosten

$$a_i := t_i + \Phi_i - \Phi_{i-1}. \tag{83}$$

Was hilft uns dies nun? Da $\Phi_0 = 0$ und $\Phi_i \geq 0$ für alle $i \geq 0$ gelten, erhalten wir

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \Big(\sum_{i=1}^{n} t_i\Big) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i. \tag{84}$$

Die Summe aller amortisierten Kosten bildet also eine obere Schranke für die Summe aller realen Kosten, oder anders ausgedrückt, die durchschnittlichen realen Kosten aller Operationen sind höchstens so gross wie die durchschnittlichen amortisierten Kosten aller Operationen. Häufig ist es so, dass die durchschnittlichen realen Kosten schwer abzuschätzen sind. Die Kunst bei der amortisierten Analyse liegt darin, eine geeignete Potentialfunktion zu finden, sodass die durchschnittlichen amortisierten Kosten leichter berechnet werden können. Üblicherweise möchte man die Potentialfunktion Φ so wählen, dass Φ_i gegenüber Φ_{i-1} leicht anwächst, wenn die realen Kosten t_i klein sind, und dass Φ_i gegenüber Φ_{i-1} stark abfällt, wenn die realen Kosten t_i sehr gross sind.

Wahl der Potentialfunktion

Für den zuvor diskutierten Multistack können wir als Potentialfunktion die Anzahl der Elemente auf dem Stack benutzen. Konkret gibt Φ_i also die Anzahl der vorhandenen Listenelemente an, nachdem die ersten i Operationen durchgeführt wurden. Wir betrachten jetzt alle möglichen Operationen, die im i-ten Schritt auftreten können, und zeigen, dass sie alle konstante amortisierte Kosten haben.

- $\textsc{Push}(x, S)$: Die realen Kosten sind $t_i = 1$, und da S exakt ein Element hinzugefügt wurde, ist $\Phi_i - \Phi_{i-1} = 1$. Folglich sind die amortisierten Kosten $a_i = 1 + 1 = 2$.
- $\textsc{Pop}(S)$: Die realen Kosten sind $t_i = 1$, und da von S exakt ein Element entfernt wurde, ist $\Phi_i - \Phi_{i-1} = -1$. Folglich sind die amortisierten Kosten $a_i = 1 - 1 = 0$.
- $\textsc{Multipop}(k, S)$: Die realen Kosten sind $t_i = k$, denn es werden k Elemente von S entfernt. Aus dem gleichen Grund fällt die Potentialfunktion um k ab, wir haben also $\Phi_i - \Phi_{i-1} = -k$. Folglich sind die amortisierten Kosten erneut $a_i = k - k = 0$.

Die amortisierten Kosten von $\textsc{Top}(S)$, $\textsc{isEmpty}(S)$ sowie $\textsc{EmptyStack}$ können analog bestimmt werden. Wir sehen also wie zuvor, dass alle Operationen nur konstante amortisierte Kosten haben. Insbesondere braucht auch $\textsc{Multipop}(k, S)$ im Durchschnitt über alle Operationen nur konstante Zeit.

Wörterbuch (Dictionary) In dieser und der folgenden Vorlesung werden wir Datenstrukturen und Algorithmen zur Implementierung eines *Wörterbuchs* kennenlernen. Dieser ADT dient zur Verwaltung von *Schlüsseln*, die aus einem geordneten Universum $\mathcal{K}$ (z.B. der Menge der natürlichen Zahlen) stammen und stellt drei wesentliche Operationen zur Verfügung:

OPERATIONEN

- INSERT(k, D): Fügt den neuen Schlüssel $k \in \mathcal{K}$ in das Wörterbuch D ein. Falls k in D bereits vorhanden ist, wird eine Fehlermeldung ausgegeben.
- DELETE(k, D): Löscht den Schlüssel $k \in \mathcal{K}$ aus dem Wörterbuch D. Falls k in D nicht vorhanden ist, wird eine Fehlermeldung ausgegeben.
- SEARCH(k, D): Liefert genau dann "true" zurück, wenn das Wörterbuch D den Schlüssel $k \in \mathcal{K}$ enthält, und "false" ansonsten.

In dieser Definition erfolgt also vor jeder Einfügeoperation eine erfolglose Suche, und vor jeder Löschoperation eine erfolgreiche Suche nach dem entsprechenden Schlüssel.

PRAKTISCHE ÜBERLEGUNGEN

Wörterbücher haben sehr viele Anwendungen in vielen praktisch relevanten Bereichen, z.B. in Datenbanken und Dateisystemen. In realen Situationen würden die obigen Operationen vermutlich leicht angepasst: So möchte man üblicherweise nicht nur einen Schlüssel (z.B. den Namen einer Person) speichern, sondern auch einen mit ihm verknüpften Datensatz (der etwa das Geburtsdatum oder die Anschrift der entsprechenden Person enthält). Auch möchte man beim Suchen oftmals den Datensatz selbst zurückgeben lassen, und nicht allein eine Information ob unter dem gegebenen Schlüssel ein entsprechender Datensatz gespeichert ist. Der Einfachheit halber ersparen wir uns im Folgenden diese technischen Details und betrachten vor allen Dingen die drei zuvor genannten Operationen. Reale Implementierungen von Wörterbüchern unterstützen darüber hinaus oftmals eine Vielzahl weiterer Operationen, etwa die Ausgabe aller Schlüssel in alphabetisch aufsteigender Reihenfolge, die Vereinigung zweier Wörterbücher, usw.

NAIVE REALISIERUNGEN

Wir haben bereits einfache Datenstrukturen kennengelernt, die zur Implementierung eines Wörterbuchs benutzt werden können. So könnte man z.B. eine einfach verkettete Liste benutzen. Eine erfolglose Suche dauert allerdings Zeit $\Theta(n)$, wenn die Liste n Elemente speichert; Einfügen ist daher nur in Zeit $\Theta(n)$ möglich. Das gleiche Problem tritt bei der Verwendung eines (hinreichend grossen) Arrays auf, in das jeweils an eine freie Position eingefügt wird. Schneller suchen, nämlich in Zeit $\mathcal{O}(\log n)$, können wir in sortierten Arrays. Wollen wir allerdings einen Schlüssel einfügen, der kleiner als alle bisher gespeicherten Schlüssel ist, müssen aufgrund der starren Struktur eines Arrays alle Schlüssel um eine Position nach hinten verschoben werden. Die Einfügung eines Schlüssels kann also im schlechtesten Fall erneut Zeit $\Theta(n)$ in Anspruch nehmen. Wir werden nun Implementierungen kennenlernen, bei denen die Wörterbuchoperationen wesentlich schneller ausgeführt werden können.

4.2 Hashing

SCHLÜSSEL-UNIVERSUM

Bei der Definition des ADT Wörterbuch hatten wir gefordert, dass die Menge der zu verwaltenden Schlüssel aus einem geordneten Universum $\mathcal{K}$ stammen. In jedem Rechner können beliebige Objekte durch eine Folge von Bits codiert werden, die wiederum als Binärzahl interpretiert werden können. Es ist also keine wesentliche Einschränkung, wenn wir in diesem Abschnitt annehmen, dass $\mathcal{K} \subseteq \mathbb{N}_0$ gilt, wir also nur natürliche Zahlen als Schlüssel zulassen.

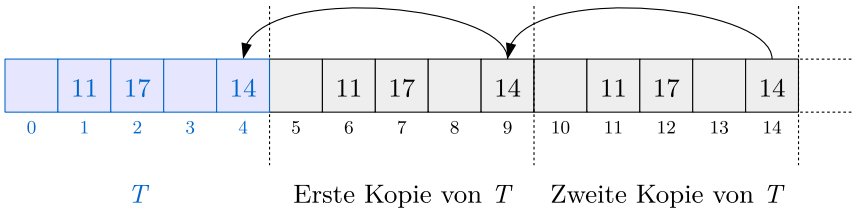


Abb. 4.4 Eine Hashtabelle T mit $m = 5$ Plätzen, in der bereits die Schlüssel 11 und 17 gespeichert sind (blau), und in die $k = 14$ eingefügt werden soll. Wir legen gedanklich zwei Kopien von T (schwarz) an T, fügen k an Position 14 ein und rechnen dies auf die Positionen $0, \ldots, m-1$ zurück, was $14 \bmod 5 = 4$ ergibt.

Speicherung im Array

Wenn der grösste zu verwaltende Schlüssel k_{max} im Vorfeld bekannt wäre, könnten wir ein Array T benutzen, das von 0 bis k_{max} indiziert ist, und könnten jeden Schlüssel k einfach an Position $T[k]$ speichern. Auf diese Art wären alle drei Wörterbuchoperationen in konstanter Zeit durchführbar. Allerdings ist diese Idee wenig praktikabel, denn sie benötigt u.U. sehr viel Platz. Zudem kann k_{max} unbekannt sein. *Hashing* wandelt diese Idee nun ab, indem die Grösse des Arrays T auf m Plätze (für eine geeignet gewählte Zahl m) beschränkt wird. Die Positionen von T seien von 0 bis $m-1$ nummeriert. Das Array T wird als *Hashtabelle* bezeichnet. Ein Schlüssel $k \in \{0, \ldots, m-1\}$ kann wie vorher an Position k gespeichert werden. Was können wir aber mit Schlüsseln $k \geq m$ tun? Dazu können wir uns gedanklich vorstellen, identisch gefüllte Kopien von T zu erzeugen und nebeneinander zu legen. Die Positionen von T selbst sind dann $0, \ldots, m-1$, die der ersten Kopie sind $m, \ldots, 2m-1$, die der zweiten Kopie $2m, \ldots, 3m-1$, usw. Wir speichern dann gedanklich den Schlüssel k an der Position k und rechnen die Position auf die entsprechende Position in T selbst zurück (siehe Abbildung 4.4). Wir beobachten, dass k an der Position $k \bmod m$ gespeichert wird. Allgemein haben wir also eine *Hashfunktion* $h : \mathcal{K} \to \{0, 1, \ldots, m-1\}$, die dem Schlüssel k seine *Hashadresse* $h(k)$ zuweist. Man beachte, dass im Allgemeinen $|\mathcal{K}| \gg m$ ist. Wir müssen also damit rechnen, dass es verschiedene Schlüssel $k, k' \in \mathcal{K}$ mit $h(k) = h(k')$ gibt, d.h., dass *Kollisionen* auftreten. Leider sind Kollisionen sogar recht wahrscheinlich, wie das folgende Theorem zeigt.

Hashing

Hashtabelle

Hashfunktion
Hashadresse

Kollision

Theorem 4.1 (Geburtstagsparadoxon)**.** *In einem Raum mit 23 Personen ist die Wahrscheinlichkeit, dass mindestens zwei dieser Personen am gleichen Tag Geburtstag haben, grösser als* 50%*.* ∎

Geburtstags-paradoxon

In Bezug auf Hashing lässt sich das Geburtstagsparadoxon wie folgt interpretieren: Ist eine Hashtabelle der Grösse $m = 365$ gegeben und verteilen sich die Schlüssel aus $\mathcal{K}$ gleichmässig auf die Positionen $0, 1, \ldots, 364$, dann ist es schon bei 23 zufällig ausgewählten Schlüsseln wahrscheinlicher, dass eine Kollision auftritt, als dass keine auftritt. Eine Hashfunktion sollte daher die Schlüssel aus $\mathcal{K}$ möglichst gleichmässig auf die Positionen der Hashtabelle verteilen. Glücklicherweise ist dies für die zuvorgenannte Hashfunktion $h(k) = k \bmod m$ der Fall, wenn m als Primzahl gewählt wird. Man spricht dann auch von der *Divisions-Rest-Methode*. Eine weitere Möglichkeit besteht in der *multiplikativen Methode* mit der Hashfunktion

Wahl der Hashfunktion

Divisions-Rest-Methode

Multiplikative Methode

$$h(k) = \left\lfloor m\left(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor\right) \right\rfloor, \tag{85}$$

wobei $\phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.61803$ das Inverse des goldenen Schnitts darstellt. Da die

Divisions-Rest-Methode aber gute Ergebnisse liefert und zudem leicht berechenbar ist, werden wir im Folgenden immer die Hashfunktion $h(k) = k \bmod m$ benutzen; dabei werde die Grösse m der Hashtabelle als Primzahl gewählt.

4.2.1 Universelles Hashing

In praktischen Anwendungen muss leider befürchtet werden, dass viele Schlüssel auf die gleiche Hashadresse abgebildet werden, selbst wenn bei die Grösse m der Hashtabelle als Primzahl gewählt und die Divisions-Rest-Methode benutzt wird. Andererseits tritt diese Situation nur bei unglücklicher Wahl der Hashfunktion auf. Hätten wir eine Menge $\mathcal{H}$ möglicher Hashfunktionen zur Verfügung und wählen wir zu Beginn des Einfügevorgangs *einmalig* eine Hashfunktion zufällig aus $\mathcal{H}$ (die dann im Folgenden für alle Einfügevorgänge stets gleich bleibt), dann führt eine "schlecht" gewählte Schlüsselmenge nicht notwendigerweise zu vielen Kollisionen.

Universelles Hashing

Eine sogenannte *universelle Hashklasse* ist eine Menge $\mathcal{H} \subseteq \{h : \mathcal{K} \to \{0, \ldots, m-1\}\}$ von Hashfunktionen, sodass

$$\forall k_1, k_2 \in \mathcal{K} \text{ mit } k_1 \neq k_2 : \frac{|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}|}{|\mathcal{H}|} \leq \frac{1}{m} \tag{86}$$

gilt. Wird nun für feste Schlüssel $k_1, k_2 \in \mathcal{K}$ die Hashfunktion h zufällig aus $\mathcal{H}$ gewählt (und bleibt über alle Einfügevorgänge gleich), dann ist die Wahrscheinlichkeit für eine Kollision nicht grösser als wenn die Hashfunktion fest und die Schlüssel k_1 und k_2 zufällig gewählt würden. Von einer Klasse von Hashfunktionen können wir also nicht mehr erwarten! Man kann nun zeigen, dass universelle Hashklassen nicht nur existieren, sondern darüber hinaus auch noch einfach konstruierbar sind:

Existenz perfekter Hashklassen

Theorem 4.2. *Seien p eine Primzahl und $\mathcal{K} = \{0, \ldots, p-1\}$. Definieren wir für $a \in \{1, \ldots, p-1\}$ und $b \in \{0, \ldots, p-1\}$ die Hashfunktion*

$$\begin{aligned} h_{ab} : \mathcal{K} &\to \{0, \ldots, m-1\} \\ k &\mapsto \big((ak + b) \bmod p\big) \bmod m, \end{aligned} \tag{87}$$

dann ist $\mathcal{H} = \{h_{ab} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$ eine universelle Hashklasse. ∎

4.2.2 Hashverfahren mit Verkettung der Überläufer

Wir haben bereits argumentiert, dass wir stets mit Kollisionen rechnen müssen. Es gibt nun verschiedene Strategien zur Kollisionsbehandlung. Eine naheliegende Variante besteht darin, an jeder Tabellenposition nicht einen einzigen Schlüssel, sondern eine Liste von Schlüsseln zu speichern, die alle die gleiche Hashadresse besitzen. Man spricht dann auch von einer *direkten Verkettung der Überläufer*. Die Wörterbuchoperationen können nun sehr leicht wie folgt implementiert werden.

Direkte Verkettung

Suchen — Search(k, D). Durchlaufe die an Tabellenposition $h(k)$ gespeicherte Liste und gib "true" zurück, falls sie ein Listenelement enthält, das k speichert. Wird das Ende der Liste erreicht, ohne dass k gefunden wird, dann ist k nicht in der Hashtabelle vorhanden, also gib "false" zurück.

Einfügen — Insert(k, D). Prüfe, ob die an der Tabellenposition $h(k)$ gespeicherte Liste den Schlüssel k enthält. Falls nein, füge k am Ende dieser Liste ein.

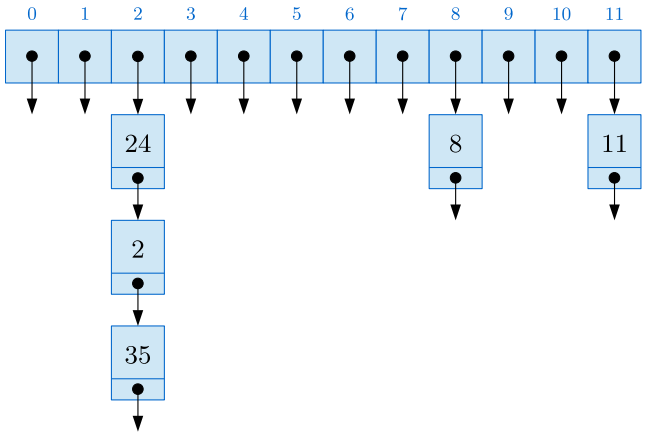


Abb. 4.5 Eine Hashtabelle T mit $m = 11$ Plätzen und direkter Verkettung der Überläufer. Eine Einfügung der Schlüssel $8, 11, 24, 2, 35$ in dieser Reihenfolge führt (neben weiteren Reihenfolgen) zur dieser Tabelle.

LÖSCHEN

DELETE(k, D). Durchlaufe die an der Tabellenposition $h(k)$ gespeicherte Liste. Wird dabei ein Listenelement gefunden, das k speichert, dann entferne dieses Listenelement.

Die Laufzeit der Operationen hängt offenbar von der Grösse der Hashtabelle und der Anzahl der gespeicherten Schlüssel ab. Hat eine Hashtabelle Grösse m und werden derzeit n Schlüssel verwaltet, dann wird $\alpha = n/m$ als *Belegungsfaktor* bezeichnet. Für eine genaue Laufzeitabschätzung beobachten wir, dass die Laufzeit des Einfügens und des Löschens durch die Zeit zur erfolglosen bzw. erfolgreichen Suche nach einem Schlüssel dominiert werden. Daher genügt es, die erwartete Anzahl betrachteter Listeneinträge

BELEGUNGSFAKTOR

- C'_n bei erfolgloser Suche, sowie
- C_n bei erfolgreicher Suche

zu analysieren. Bei einer erfolglosen Suche nach einem Schlüssel k müssen alle Elemente der an Position $h(k)$ gespeicherten Liste betrachtet werden. Die Anzahl betrachteter Elemente ist gleich der Länge der entsprechenden Liste. Da die durchschnittliche Listenlänge $\alpha = n/m$ beträgt, erhalten wir

ERFOLGLOSE SUCHE

$$C'_n = \alpha. \tag{88}$$

Ist die Suche dagegen erfolgreich und befindet sich der gesuchte Schlüssel k im i-ten Eintrag in der entsprechenden Liste, dann werden genau i Listeneinträge betrachtet. Um den Erwartungswert von i zu bestimmen, analysieren wir die Zeit zur erfolgreichen Suche des j-ten eingefügten Schlüssels. Vereinfachend nehmen wir an, dass Schlüssel grundsätzlich ans Listenende eingefügt werden, und dass niemals Schlüssel gelöscht wurden. Vor dem Einfügen des j-ten Schlüssels betrug die durchschnittliche Listenlänge $(j-1)/m$. Also betrachtet die erfolgreiche Suche nach diesem Schlüssel im Erwartungswert $1 + (j-1)/m$ Listeneinträge. Für jedes $j \in \{1, \ldots, n\}$ beträgt die Wahrscheinlichkeit, dass k als j-tes eingefügt wurde, genau $1/n$. Also ist

ERFOLGREICHE SUCHE

$$C_n = \frac{1}{n} \sum_{j=1}^{n} \big(1 + (j-1)/m\big) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}. \tag{89}$$

VERBESSERUNG

Das Verfahren hat noch Potential zur Verbesserung. Wir haben der Einfachheit halber angenommen, dass ein Schlüssel immer ans Ende der entsprechenden Liste eingefügt wird. Halten wir dagegen die Listen sortiert (d.h., fügen wir einen Schlüssel gleich an der korrekten Position der sortierten Reihenfolge ein), dann verbessert dies die Zeit zur erfolglosen Suche. Die Zeit zur erfolgreichen Suche verschlechtert sich nicht, lediglich ihre Analyse wird komplizierter.

VOR- UND NACHTEILE

Vorteile der Strategie sind, dass Belegungsfaktoren $\alpha > 1$ problemlos möglich sind, d.h., dass mehr Schlüssel verwaltet werden können als die Tabelle Plätze hat. Ausserdem ist das Entfernen von Schlüsseln unkompliziert und ohne negative Konsequenzen möglich. Negativ ist der erhöhte Speicherplatzbedarf, denn für jeden Listeneintrag muss ein Zeiger auf den nachfolgenden Eintrag gespeichert werden.

4.2.3 Offenes Hashing

Anstatt die Überläufer in einer Liste zu verwalten, können wir sie auch direkt in der Hashtabelle speichern. Auf diese Weise entfällt der für die Listenzeiger benötigte Extraplatz. Um bei diesem *offenen* Hashing die Position der Überläufer zu finden,

SONDIERUNGSFUNKTION

benutzen wir eine *Sondierungsfunktion* $s(j,k)$, wobei $j \in \{0,\ldots,m-1\}$ und $k \in \mathcal{K}$ ein Schlüssel ist. Zur Bestimmung der Tabellenposition eines Schlüssels k betrachten wir die Tabellenpositionen entlang der *Sondierungsfolge*

SONDIERUNGSFOLGE

$$\big(h(k) - s(0,k)\big) \bmod m, \ldots, \big(h(k) - s(m-1,k)\big) \bmod m. \tag{90}$$

Speichert ein Eintrag $\big(h(k) - s(j,k)\big) \bmod m$ den Schlüssel k, dann haben wir die Position von k gefunden und sind fertig (die übrigen Positionen der Folge müssen dann nicht mehr betrachtet werden). Ist der Eintrag frei, dann ist der Schlüssel k in der Tabelle nicht vorhanden und wir sind ebenfalls fertig. Weitersuchen müssen wir lediglich, wenn der Eintrag einen Schlüssel ungleich k enthält. In diesem Fall fahren wir analog mit dem Tabelleneintrag $\big(h(k) - s(j+1,k)\big) \bmod m$ fort. Die Wörterbuchoperationen können nun wie folgt implementiert werden.

SUCHEN

SEARCH(k, D). Betrachte die Tabelleneinträge gemäss der Sondierungsfolge (90). Speichert ein Tabelleneintrag den Schlüssel k, gib "true" zurück und brich die Suche ab. Wird ein leerer Tabelleneintrag gefunden oder ist k in der gesamten Sondierungsfolge nicht vorhanden, gib "false" zurück.

EINFÜGEN

INSERT(k, D). Führe eine Suche nach dem Schlüssel k durch. Ist der Schlüssel k nicht vorhanden, füge k an die erste freie Position der Sondierungsfolge ein.

LÖSCHEN

DELETE(k, D). Führe eine Suche nach dem Schlüssel k durch. Falls k gefunden wird, markiere die Position von k mit einem speziellen Gelöscht-Flag. Die Position darf nicht einfach als frei markiert werden, da ansonsten die Suche nach einem Schlüssel nicht mehr funktioniert (werden z.B. viele Schlüssel $k_1, \ldots, k_l$ mit der gleichen Hashadresse eingefügt und wird dann k_1 gelöscht, dann wäre die erste Position der Sondierungsfolge ein freier Tabellenplatz und die Suche würde direkt terminieren). Das Gelöscht-Flag kann z.B. durch einen speziellen Schlüssel repräsentiert werden, der dann natürlich nicht mehr regulär eingefügt werden darf. Wird bei der Suche ein als gelöscht markiertes Feld gefunden, muss die Suche regulär mit der nächsten Position der Sondierungsfolge fortgesetzt werden. Neue Schlüssel dürfen aber natürlich auf Positionen eingefügt werden, die als gelöscht markiert sind.

Abb. 4.6 Eine offene Hashtabelle T mit $m = 11$ Plätzen, bei der lineares Sondieren zur Kollisionsauflösung benutzt wird. Die Schlüssel $8, 11, 24, 2, 35$ wurden in dieser Reihenfolge eingefügt.

Nachdem wir die Realisierung der Wörterbuchoperationen beschrieben haben, müssen wir nun überlegen, was geeignete Sondierungsfunktionen sind. Auf jeden Fall sollte die Sondierungsfolge (90) alle Tabellenpositionen berücksichtigen, d.h., sie sollte eine Permutation von $\{0, \ldots, m-1\}$ sein. Wir beschreiben nun einige Sondierungsstrategien, die diese Eigenschaft besitzen.

Lineares Sondieren

Lineares Sondieren: $s(j,k) = j$**.** Die zugehörige Sondierungsfolge ist

$$h(k) \bmod m, \big(h(k)-1\big) \bmod m, \ldots, \big(h(k)-m+1\big) \equiv \big(h(k)+1\big) \bmod m, \quad (91)$$

d.h., ausgehend von der Position $h(k)$ laufen wir in der Tabelle so lange nach links, bis eine freie Position gefunden wird. Wird die Position ganz links (mit Index 0) erreicht, fahren wir bei der Position ganz rechts (mit Index $m-1$) fort. Diese Sondierungsstrategie hat den Nachteil, dass auch bei geringen Belegunsfaktoren α schnell lange zusammenhängende Blöcke von belegten Tabellenplätzen entstehen. Um dies einzusehen, nehmen wir an, alle Tabellenpositionen i innerhalb eines Intervalls $[i_l, i_u] \subseteq \{0, \ldots, m\}$ wären bereits belegt. Wird nun ein Schlüssel k mit Hashadresse $h(k) \in [i_l, i_u]$ eingefügt, dann wird dieser direkt links vom genannten Bereich gespeichert (wenn $i_l > 0$ ist; für $i_l = 0$ wird der Schlüssel entsprechend möglichst weit rechts in der Tabelle gespeichert). Aus diesem Grund wachsen bereits bestehende lange Blöcke schneller als kurze Blöcke. Diese Eigenschaft führt auch zu langen Suchzeiten: Man kann zeigen, dass die erfolglose bzw. erfolgreiche Suche

$$C'_n \approx \frac{1}{2}\Big(1 + \frac{1}{(1-\alpha)^2}\Big) \quad \text{bzw.} \quad C_n \approx \frac{1}{2}\Big(1 + \frac{1}{1-\alpha}\Big) \quad (92)$$

viele Tabellenpositionen betrachten. Dies zeigt, wie ineffizient lineares Sondieren bei hohen Belegungsfaktoren ist: Für $\alpha = 0.95$ betrachtet eine erfolglose Suche im Durchschnitt mehr als 200 Tabelleneinträge!

Quadratisches Sondieren

Quadratisches Sondieren: $s(j,k) = (\lceil j/2 \rceil)^2 (-1)^{j+1}$**.** Ein Problem beim linearen Sondieren besteht darin, dass ähnliche Hashadressen auch ähnliche Sondierungsfolgen besitzen, was wie gesehen schnell zu langen zusammenhängenden Bereichen von belegten Positionen führt. Quadratisches Sondieren vermeidet dieses Problem, indem die Abstände zwischen der ursprünglichen Hashadresse und der entsprechenden Adresse in der Sondierungsfolge quadratisch wachsen. Zur oben angegebenen Sondierungsfunktion gehört die Sondierungsfolge

$$\begin{aligned} &h(k) \bmod m, \big(h(k)-1\big) \bmod m, \big(h(k)+1\big) \bmod m, \\ &\quad \big(h(k)-4\big) \bmod m, \big(h(k)+4\big) \bmod m, \big(h(k)-9\big) \bmod m, \ldots \end{aligned} \quad (93)$$

Man kann zeigen, dass (93) eine Permutation aller Tabellenpositionen $\{0, \ldots, m-1\}$

Abb. 4.7 Eine offene Hashtabelle T mit $m = 11$ Plätzen, bei der quadratisches Sondieren zur Kollisionsauflösung benutzt wird. Die Schlüssel $8, 11, 24, 2, 35$ wurden in dieser Reihenfolge eingefügt.

Abb. 4.8 Eine offene Hashtabelle T mit $m = 11$ Plätzen, bei der Double Hashing zur Kollisionsauflösung benutzt wird. Die Schlüssel $8, 11, 24, 2, 35$ wurden in dieser Reihenfolge eingefügt.

bildet, wenn m eine Primzahl ist und $m \equiv 3 \bmod 4$ erfüllt. Die erfolglose bzw. erfolgreiche Suche betrachten

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right) \quad \text{bzw.} \quad C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \tag{94}$$

viele Tabellenpositionen. Insbesondere betrachtet die erfolglose Suche bei einem Belegungsfaktor von $\alpha = 0.95$ nur ca. 22 Tabelleneinträge, was eine enorme Verbesserung gegenüber linearem Sondieren darstellt. Trotzdem hat diese Sondierungsstrategie einen Nachteil: Alle Schlüssel mit gleichen Hashadressen besitzen die gleiche Sondierungsfolge. Werden nun viele Schlüssel mit der gleichen Hashadresse eingefügt, dann führt dies zu vielen sogenannten *Sekundärkollisionen* entlang der Sondierungsfolge. Wir werden im Folgenden eine Strategie kennenlernen, bei der auch Schlüssel mit gleichen Hashadressen unterschiedliche Sondierungsfolgen haben können.

Sekundärkollision

Double Hashing

Double Hashing Sei $h'(k)$ eine zweite, von $h(k)$ unabhängige Hashfunktion. *Double Hashing* besitzt die Sondierungsfunktion $s(j,k) = j \cdot h'(k)$ mit der zugehörigen Sondierungsfolge

$$h(k) \bmod m, \big(h(k) - h'(k)\big) \bmod m, \big(h(k) - 2h'(k)\big) \bmod m, \ldots \tag{95}$$

Die Wahl der zweiten Hashfunktion $h'(k)$ ist nicht trivial. Offenbar darf sie m nicht teilen und muss für alle Schlüssel k ungleich 0 sein. Man kann aber zeigen, dass alle genannten Anforderungen erfüllt werden, wenn m eine Primzahl ist und $h(k) = k \bmod m$ sowie $h'(k) = 1 + (k \bmod (m-2))$ gewählt werden. In diesem Fall betrachten die erfolglose bzw. erfolgreiche Suche

$$C'_n \approx \frac{1}{1-\alpha} \quad \text{bzw.} \quad C_n \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \ldots < 2.5 \tag{96}$$

viele Tabellenpositionen.

Cuckoo Hashing

Cuckoo Hashing Kuckucks-Hashing (engl. *Cuckoo Hashing*) benutzt zwei Hashtabellen T_1 und T_2 mit zwei Hashfunktionen $h_1(k)$ und $h_2(k)$. Um nach einem Schlüssel k zu suchen, schauen wir zunächst an Position $h_1(k)$ in T_1 nach. Ist k dort nicht gespeichert, dann prüfen wir, ob k an der Position $h_2(k)$ in T_2 gespeichert ist. Ist dies ebenfalls nicht der Fall, dann ist k nicht vorhanden.

Das Einfügen wird jetzt aber komplizierter. Wir fügen einen Schlüssel k an die Position $h_1(k)$ in T_1 ein. War dort vorher kein Schlüssel gespeichert, dann sind wir fertig. Ansonsten wird der vorher dort gespeicherte Schlüssel k_1 verdrängt. Konkret fügen wir k_1 (nicht k!) mit h_2 in T_2 ein. War dort vorher ein Schlüssel k_2 gespeichert, dann wird dieser mit h_1 in T_1 eingefügt. Der ggf. dort gespeicherte Schlüssel k_3 wird dann mit h_2 in T_2 eingefügt. Auf diese Art verfährt man weiter, bis entweder eine freie Position gefunden wird, oder man einen Kreis erhält (wenn der ursprünglich einzufügende Schlüssel k aus T_2 verdrängt wird und in T_1 eingefügt werden soll). Wenn ein solcher Kreis entsteht, dann erzeugt man zwei neue, grössere Hashtabellen, wählt zwei neue Hashfunktionen, fügt alle vorher gespeicherten Schlüssel (zuzüglich k) auf genau die gleiche Art und Weise in die neuen Hashtabellen ein und verwirft die alten Hashtabellen. Man kann zeigen, dass dies im Erwartungswert auch nur konstante Zeit benötigt.

Einfügen

4.3 Selbstanordnung

Wir haben bereits überlegt, wie einfach verkettete Listen zur Realisierung von Wörterbüchern benutzt werden können. Problematisch war hier vor allen Dingen eine im schlechtesten Fall lineare Suchzeit. Angenommen, die Liste speicherte die Schlüssel $k_1, \ldots, k_n$, und für jeden Schlüssel k_i wüssten wir bereits, mit welcher Häufigkeit H_i auf ihn zugegriffen wird. In diesem Fall könnten wir die Schlüssel absteigend nach ihrer Häufigkeit H_i sortieren, im ersten Element der Liste den Schlüssel mit der grössten Häufigkeit speichern, im zweiten Element den Schlüssel mit der zweitgrössten Häufigkeit, usw. Auf diese Art würde sich zwar die lineare Laufzeit für die Suche im schlechtesten Fall nicht ändern, zumindest die Suche nach den häufigsten Elementen wäre aber u.U. wesentlich schneller.

In der Realität sind die Suchhäufigkeiten oftmals im Vorfeld nicht bekannt. In diesem Fall gibt es verschiedene Möglichkeiten, die Liste dynamisch so umzuordnen, dass spätere Zugriffe auf Elemente hoffentlich schneller möglich sind.

1) *Frequency Count:* Wir zählen für jeden Schlüssel k_i die bisher beobachteten Häufigkeiten H_i' mit (initial $H_i' = 0$), und erhöhen bei jeder Suche nach k_i die entsprechende Häufigkeit H_i' um 1. Die Listenelemente werden dann nach jeder Suche nach Häufigkeit absteigend sortiert angeordnet. — Frequency Count

2) *Transpose:* Bei jedem Zugriff auf einen Schlüssel k_i bewegen wir ihn genau eine Position nach vorn (sofern er nicht bereits am Listenanfang stand). — Transpose

3) *Move-to-Front:* Bei jedem Zugriff auf einen Schlüssel k_i bewegen wir ihn an den Anfang der Liste. — Move-to-Front

Wie gut sind diese Strategien? Man sieht leicht, dass Transpose nur ungenügende Ergebnisse liefert: Angenommen, eine Liste speichert die Schlüssel $k_1, k_2, \ldots, k_n$ in exakt dieser Reihenfolge und wir greifen abwechselnd auf k_n und auf k_{n-1} zu. Jeder Zugriff auf k_n verschiebt dieses Element um eine Position nach vorn, während der nachfolgende Zugriff auf k_{n-1} die ursprüngliche Ordnung der Liste wiederherstellt. Hat die Liste also n Elemente und führen wir n Zugriffe wie soeben beschrieben durch, führt dies zu einer Laufzeit von $\Theta(n^2)$. Move-to-Front dagegen wäre in dieser Situation wesentlich besser, da nur der erste Zugriff auf k_n und auf k_{n-1} jeweils Zeit $\Theta(n)$ brauchen; jeder weitere Zugriff ist dann in konstanter Zeit möglich, da sich das gesuchte Element stets an der zweiten Stelle der Liste befindet.

Analyse Transpose

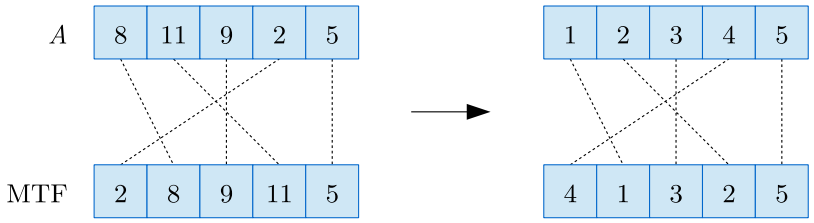


Abb. 4.9 Zwei Listen mit jeweils fünf Elementen. Gedanklich können die Elemente in der Liste von A in $1, \ldots, 5$ umbezeichnet werden; werden die Elemente von der Liste von MTF entsprechend umbezeichnet, lässt dies die Anzahl der Inversionen unverändert. Im obigen Beispiel gibt es genau vier Inversionen.

Analyse Move-to-Front

Natürlich können wir auch für Move-to-Front (im Folgenden MTF genannt) eine Folge angeben, die quadratische Zugriffszeiten benötigt. Dazu greifen wir stets auf das aktuell letzte Listenelement zu. Andererseits ist bei so einer Folge von Zugriffen nicht klar, wie eine geeignete Strategie aussehen könnte, die mit viel weniger Zugriffen auskommt. Daher vergleichen jetzt MTF mit einem idealen Konkurrenten (Algorithmus) A, um zu untersuchen, wie viel besser A werden kann. Dabei nehmen wir an, dass unser Konkurrent A so wie MTF nur das Element x bewegen darf, auf das gerade zugegriffen wurde, die anderen Elemente also nicht vertauscht. Die konkurrierende Strategie A könnte x aber an eine völlig andere Stelle als MTF bewegen. Wir wollen jetzt argumentieren, dass dies nicht (viel) besser als MTF sein kann.

Greift ein Algorithmus (entweder MTF oder A) auf ein Element x zu, das sich an der i-ten Position befindet, dann fallen allein für den Zugriff Kosten i an (denn es müssen i Listenelemente angeschaut werden). Wenn die entsprechende Strategie das Element x jetzt nach vorn bewegt, fallen keine weiteren Kosten an (denn auf die Elemente vor x wurde bereits zuvor zugegriffen). Bewegt die Strategie das Element x aber nach hinten, dann verursacht dies zusätzliche Kosten 1 für jede Position, um die x weiter nach hinten bewegt wird.

Sowohl MTF als auch die optimale Strategie A starten beide mit der selben Liste, und es folgen m Zugriffe auf irgendwelche Elemente. Bei jedem dieser Zugriffe können sowohl MTF als auch A die Liste umordnen. Wir betrachten jetzt den l-ten Zugriff auf ein Element x, und wollen zunächst die amortisierten Kosten a_l^{MTF} für MTF bestimmen. Die Potentialfunktion misst nach jedem Schritt die Anzahl der Inversionen von A gegen MTF. Dies sind die Anzahl der Paare zweier Elemente x und y, deren Reihenfolgen sich in den Listen von MTF und A unterscheiden (d.h., entweder kommt x in der Liste von A vor y und in der Liste von MTF hinter y, oder umgekehrt). Schreiben wir die Elemente der Listen von MTF und A von links nach rechts auf und verbinden wir die entsprechenden Elemente durch eine Linie, dann ist die Anzahl der Inversionen exakt die Anzahl der Kreuzungspunkte zweier Linien. Abbildung 4.9 zeigt ein Beispiel. Die Abbildung zeigt auch, dass wir die Elemente der Liste von A gedanklich in $1, \ldots, n$ umbenennen können, wobei das i-te Element die Nummer i bekomme. Bezeichnen wir das jeweils entsprechende Element in der Liste von MTF gleich um, dann erhöht dies die Anzahl der Inversionen natürlich nicht, denn wir haben ja nur die Elemente umbezeichnet, nicht umgeordnet.

Angenommen, im l-ten Schritt wird nun auf das Element i zugegriffen. Dieses befindet sich aufgrund der soeben vorgenommenen Umbezeichnung der Elemente an der Position i in der Liste von A, folglich kostet A der Zugriff auf i genau $C_l^A = i$. Sei nun k die Position, an der sich i in der Liste von MTF befindet. Die Zugriffskosten

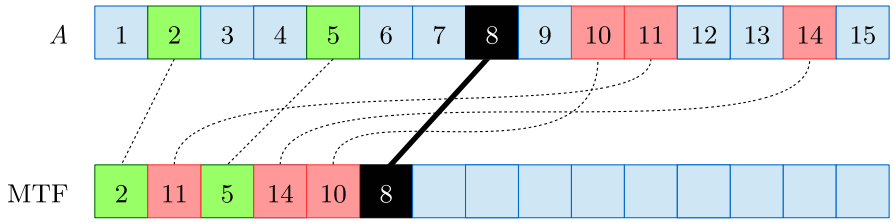


Abb. 4.10 Zugriff auf das Element $i = 8$ (schwarz), was in A Kosten $C_l^A = 8$ und in MTF Kosten $C_l^{MTF} = k = 6$ verursacht. Es gibt $x_i = 3$ Elemente, die in der Liste von MTF vor i und in der Liste von A hinter i stehen (rot eingezeichnet). Also gibt es $k - 1 - x_i$=2 Elemente, die sowohl in der Liste von A als auch in der Liste von MTF vor i stehen (grün eingezeichnet).

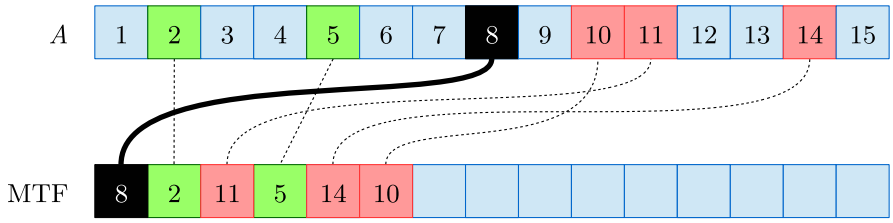


Abb. 4.11 Wird i an den Beginn der Liste von MTF bewegt, dann lösen sich die x_i Inversionen mit den roten Elementen auf, dafür kommen $k - 1 - x_i$ viele Inversionen mit den grünen Elementen neu hinzu.

für MTF auf das Element i sind dann $C_l^{MTF} = k$. Wie verändert sich nun die Potentialfunktion, wenn sowohl MTF als auch A auf i zugreifen? Es sei x_i die Anzahl der Elemente, die in der Liste von MTF vor i und in der Liste von A hinter i liegen. Da in der Liste von MTF genau $k - 1$ Elemente vor i liegen, muss es also genau $k - 1 - x_i$ viele Elemente geben, die sowohl in der Liste von MTF als auch in der Liste von A vor i liegen (siehe Abbildung 4.10). Führen wir gedanklich zuerst den Zugriff auf i in MTF durch, dann ziehen wir i an den Beginn der Liste. Dadurch verlieren wir x_i Inversionen mit den Elementen, die in der Liste von A hinter i liegen, gewinnen aber $k - 1 - x_i$ neue Inversionen mit den Elementen, die in der Liste von A vor i liegen, hinzu (siehe Abbildung 4.11). Jetzt führt A den Zugriff auf i durch. Wir beobachten nun, dass für jede Position, um die i nach vorne bewegt wird, eine Inversion wegfällt. Analog erhalten wir pro Position, um die i nach hinten bewegt wird, eine neue Inversion hinzu. Die Anzahl der Inversionen steigt also nur dann, wenn A das Element i nach hinten bewegt. Sei X_l^A die Anzahl der kostenpflichtigen Vertauschungen, d.h., die Anzahl der Stellen, um die A das Element im l-ten Schritt nach hinten bewegt (wird das Element im l-ten Schritt nach vorne bewegt, dann ist $X_l^A = 0$). Wir schätzen die Potentialdifferenz zwischen dem l-ten und dem $(l-1)$-ten Schritt durch

$$\Phi_l - \Phi_{l-1} \leq -x_i + (k - 1 - x_i) + X_l^A, \tag{97}$$

nach oben ab, d.h., Φ_l ist gegenüber Φ_{l-1} um höchstens diesen Wert gewachsen. Für die amortisierten Kosten von MTF im l-ten Schritt ergibt sich damit

$$\begin{aligned} a_l^{MTF} &= C_l^{MTF} + \Phi_l - \Phi_{l-1} && (98)\\ &\leq k - x_i + (k - 1 - x_i) + X_l^A && (99)\\ &= (k - x_i) + (k - x_i - 1) + X_l^A \leq C_l^A + C_l^A - 1 + X_l^A, && (100) \end{aligned}$$

denn die realen Kosten C_l^A von A beim Zugriff auf i sind mindestens $k - x_i$ (da mindestens so viele Elemente in der Liste von A vor i kommen). Haben wir nun eine Folge von m Zugriffen, dann hat MTF Kosten $\sum_{l=1}^{n} C_l^{MTF}$ und A Kosten $\sum_{l=1}^{n}(C_l^A + X_l^A)$. Der Summand X_l^{MTF} entfällt in der Summe der Kosten von MTF, da er stets 0 ist (MTF bewegt ein Element stets nach vorne, niemals nach hinten). Da nun die realen Kosten von MTF durch die amortisierten Kosten nach oben abgeschätzt werden können (die Potentialfunktion ist initial 0, da die Listen gleich sind, und wird niemals negativ), folgt

$$\sum_{l=1}^{n} C_l^{MTF} \leq \sum_{l=1}^{n} a_l^{MTF} \leq \sum_{l=1}^{n}(2C_l^A - 1 + X_l^A) \tag{101}$$

$$\leq \sum_{l=1}^{n} 2(C_l^A + X_l^A) \leq 2\sum_{l=1}^{n}(C_l^A + X_l^A). \tag{102}$$

Move-to-Front führt damit also selbst im schlechtesten Fall höchstens doppelt so viele Operationen wie eine optimale Strategie aus.

4.4 Natürliche Suchbäume

Motivation

Wir haben mit Hashing bereits eine Implementierung von Wörterbüchern kennengelernt, die im besten sowie im durchschnittlichen Fall sehr schnelle Zugriffszeiten hat. Auch haben wir überlegt, wie man erreichen kann, dass man den schlechtesten Fall mit hoher Wahrscheinlichkeit umgehen kann. Schlimmer als eine lineare Suchzeit im schlechtesten Fall ist jedoch, dass Hashing manche Anfragen überhaupt nicht unterstützt. Bisher haben wir nur das Suchen, Einfügen und Entfernen von Schlüsseln diskutiert. In vielen realen Anwendungen möchte man aber beispielsweise auch die gespeicherten Schlüssel in aufsteigend sortierter Reihenfolge ausgeben können. Ebenso möchte man vielleicht für einen Schlüssel k, der nicht im Wörterbuch enthalten ist, den nächstkleineren Schlüssel k' finden, der im Wörterbuch enthalten ist. Beide Operationen können mit Hashing nicht implementiert werden.

Binärer Baum

Wir haben bereits früher das Konzept eines binären Baums kennengelernt (z.B. bei der unteren Schranke für allgemeine Sortierverfahren und bei Heaps). Ein *binärer Baum*

- ist entweder ein *Blatt*, d.h. der Baum ist leer, oder er
- besteht aus einem *innerer Knoten* v mit zwei Bäumen $T_l(v)$ und $T_r(v)$ als linkem bzw. rechtem Nachfolger. Der Baum $T_l(v)$ heisst *linker Teilbaum von* v, $T_r(v)$ heisst *rechter Teilbaum von* v.

Wurzel

Jeder Baum besitzt genau einen Knoten ohne Vorgänger, die sog. *Wurzel*, die in den folgenden Algorithmenbeschreibungen Root genannt wird. In jedem inneren Knoten v speichern wir

- einen Schlüssel v.Key,
- einen Zeiger v.Left auf den linken Nachfolgerknoten (also auf die Wurzel des linken Teilbaums $T_l(v)$ und *nicht* auf den Teilbaum als ganzes), sowie
- einen Zeiger v.Right auf den rechten Nachfolgerknoten (also auf die Wurzel des rechten Teilbaums $T_r(v)$).

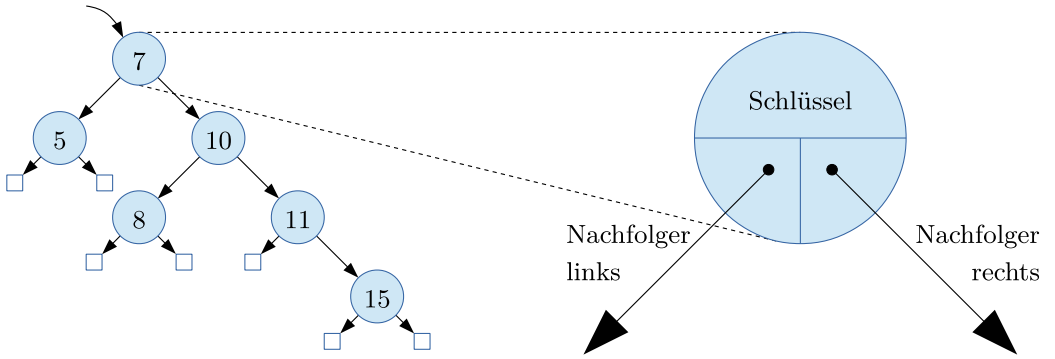


Abb. 4.12 Ein möglicher binärer Suchbaum zur Schlüsselmenge $\{5, 7, 8, 10, 11, 15\}$. Innere Knoten werden durch Kreise, Blätter durch Rechtecke dargestellt. Die Wurzel ist der Knoten 7 mit den Nachfolgerknoten 5 und 10.

Ist der linke (bzw. rechte) Nachfolger eines inneren Knotens ein Blatt, dann setzen wir v.Left (bzw. v.Right) auf **null**. Ein Zeiger auf die Wurzel genügt dann zur Repräsentation des gesamten Baums. Der Baum wird also vollständig durch Zeiger auf die entsprechenden Nachfolger repräsentiert, und nicht z.B. als Array (wie bei Heaps).

Trotzdem ist noch nicht klar, wie dies bei der Suche nach einem gegebenen Schlüssel hilft. Ein Heap könnte z.B. auch durch entsprechende Zeiger repräsentiert werden, aber es ist unklar, wie man dort effizient suchen kann. Daher führen wir nun ein weiteres Kriterium ein: Ein *binärer Suchbaum* ist ein binärer Baum, der die *Suchbaumeigenschaft* erfüllt: Jeder innere Knoten v speichert einen Schlüssel k, alle im linken Teilbaum $T_l(v)$ von v gespeicherten Schlüssel sind kleiner als k und alle im rechten Teilbaum $T_r(v)$ von v gespeicherten Schlüssel sind grösser als k.

Binärer Suchbaum

Suchbaum-eigenschaft

Suchen eines Schlüssels Abbildung 4.12 zeigt einen möglichen binären Suchbaum zur Schlüsselmenge $\{5, 7, 8, 10, 11, 15\}$. Man beachte, dass es im Allgemeinen *sehr viele verschiedene* Suchbäume gibt, die die gleiche Schlüsselmenge repräsentieren. So kann etwa jeder Schlüssel an der Wurzel stehen. Da aber alle Suchbäume die Suchbaumeigenschaft erfüllen, können wir eine binäre Suche simulieren, um nach einem gegebenen Schlüssel k zu suchen. Dazu starten wir an der Wurzel und prüfen, ob k dort gespeichert ist. Falls ja, dann haben wir k gefunden und beenden das Verfahren. Falls nein, dann fahren wir im linken Teilbaum von v fort, falls k kleiner als der Schlüssel der Wurzel ist, und ansonsten im rechten Teilbaum. Stossen wir auf ein Blatt, dann ist der Schlüssel k nicht vorhanden.

Schlüssel suchen

Search(k)

```
1 v ← Root
2 while v ist kein Blatt do
3     if k = v.Key then return true          ▷ Element gefunden
4     else if k < v.Key then v ← v.Left      ▷ Suche links weiter
6     else v ← v.Right                       ▷ Suche rechts weiter
7 return false                               ▷ k nicht gefunden
```

Zur Analyse der Laufzeit definieren wir die *Höhe eines Baums* T. Ist T ein Blatt,

Höhe

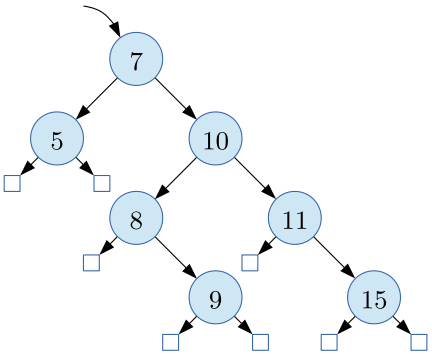


Abb. 4.13 Der Suchbaum aus Abbildung 4.12 nach der Einfügung des Schlüssels 9.

dann ist die Höhe $h(T) = 0$. Ist T ein Baum mit Wurzel v und den Teilbäumen $T_l(v)$ bzw. $T_r(v)$, dann ist

$$h(T) = 1 + \max\{h(T_l(v)), h(T_r(v))\}. \tag{103}$$

Die Höhe gibt anschaulich an, aus wie vielen Ebenen der Baum besteht. Der in Abbildung 4.12 dargestellte Baum etwa hat Höhe 4.

LAUFZEIT Es ist nun leicht zu sehen, dass die zuvor vorgestellte Suche im schlechtesten Fall Zeit $\mathcal{O}(h)$ braucht, wenn h die Höhe des binären Suchbaums ist: Die Schritte 1 sowie 3–7 benötigen nur konstante Zeit. Jeder Weg von der Wurzel zu einem Blatt hat maximal Länge h, also wird die Schleife im zweiten Schritt maximal h Mal durchlaufen.

Einfügen eines Schlüssels Beim Einfügen eines Schlüssels k führen wir zunächst eine Suche nach k durch. Wird k gefunden, dann wird der Schlüssel nicht erneut eingefügt und eine Fehlermeldung ausgegeben. Ansonsten endet die Suche erfolglos in einem Blatt. Dieses wird durch einen inneren Knoten mit dem Schlüssel k und zwei Blättern ersetzt. Fügen wir beispielsweise den Schlüssel 9 in den Suchbaum aus Abbildung 4.12 ein, dann besuchen wir die Schlüssel 7, 10 und 8, und fügen 9 als rechten Nachfolgerknoten von 8 ein.

Da zum Einfügen im Wesentlichen nur eine Suche durchgeführt wird und die anschliessende Ersetzung eines Knotens nur Zeit $\mathcal{O}(1)$ kostet, kann in einen Suchbaum der Höhe h in Zeit $\mathcal{O}(h)$ eingefügt werden.

Entfernen eines Schlüssels Es verbleibt zu zeigen, wie ein Schlüssel k aus einem binären Suchbaum entfernt werden kann. Sei dazu v der Knoten, in dem k gespeichert ist. O.B.d.A. sei v nicht die Wurzel des Baums und u der Vorgänger von v. Wir unterscheiden drei Fälle.

1. **Fall:** *Beide Nachfolger von v sind Blätter.* Dann kann der Knoten v direkt gelöscht werden, d.h. der entsprechende Nachfolger von u wird durch ein Blatt ersetzt (siehe Abbildung 4.14).

2. **Fall:** *Genau ein Nachfolger von v ist ein Blatt.* Sei w der innere Knoten, der der Nachfolger von v ist. Der entsprechende Nachfolger von u wird durch w ersetzt und v gelöscht (siehe Abbildung 4.15).

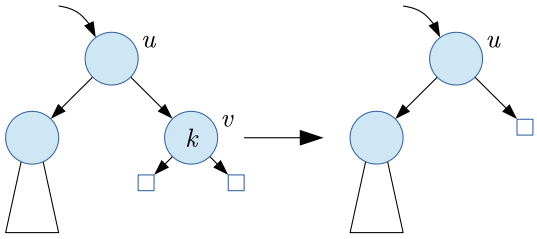


Abb. 4.14 Entfernen, Fall 1 (beide Nachfolger von v sind Blätter).

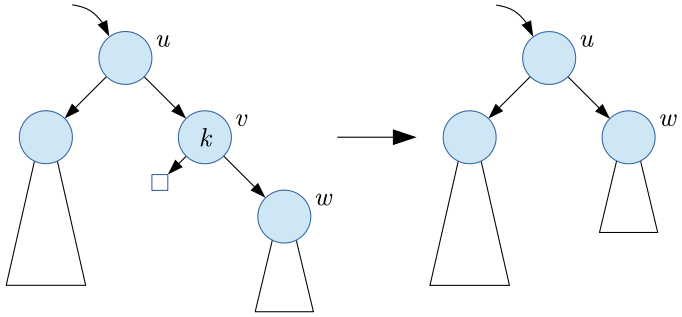


Abb. 4.15 Entfernen, Fall 2 (genau ein Nachfolger von v ist ein Blatt). Der Fall, in dem w der linke Nachfolger von v ist, wird analog aufgelöst.

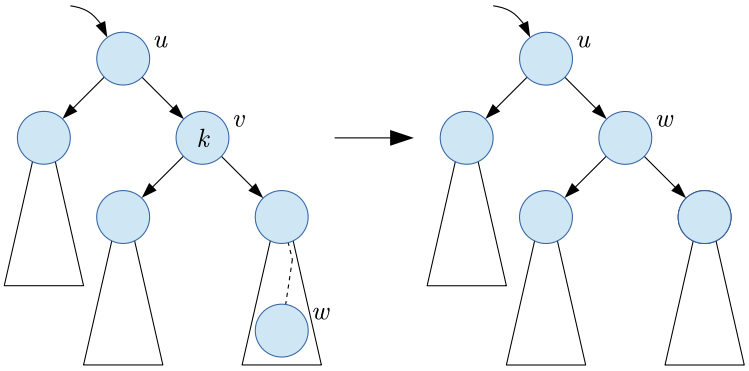


Abb. 4.16 Entfernen, Fall 3 (kein Nachfolger von v ist ein Blatt).

3. Fall: *Kein Nachfolger von v ist ein Blatt.* Dann enthalten sowohl der linke Teilbaum $T_l(v)$ als auch der rechte Teilbaum $T_r(v)$ mindestens einen inneren Knoten. Sei w der Knoten mit minimalem Schlüssel in $T_r(v)$ (dieser heisst *symmetrischer Nachfolger* von v). Wird der in v gespeicherte Schlüssel durch den in w gespeicherten Schlüssel ersetzt und anschliessend w gelöscht (siehe Abbildung 4.16), dann bleibt die Suchbaumeigenschaft erhalten (denn der Schlüssel in w ist minimal unter allen in $T_r(v)$ gespeicherten Schlüsseln). Der Knoten w hat keinen linken Nachfolgerknoten, denn dort müsste ein Schlüssel von kleinerem Wert gespeichert sein (und der Schlüssel von w wäre nicht minimal in $T_r(v)$). Also tritt beim Löschen von w nur einer der beiden erstgenannten Fälle auf.

SYMMETRISCHER NACHFOLGER

Zu einem gegebenen Knoten v kann der symmetrische Nachfolger gefunden werden, indem man genau einmal nach rechts läuft und danach so lange dem linken Nachfolger folgt, bis dieser ein Blatt als linken Nachfolger besitzt. Dies führt zu folgendem Algorithmus.

SymmetricSuccessor(v)

```
1 w ← v.Right                          ▷ Gehe genau einmal nach rechts
2 x ← w.Left                           ▷ Folge danach dem linken Nachfolger
3 while x ist kein Blatt do w ← x; x ← w.Left
4 return w                             ▷ w ist symmetrischer Nachfolger von v
```

Symmetrischer Vorgänger

Natürlich könnte auch der *symmetrische Vorgänger* (der Knoten mit grösstem Schlüssel in $T_l(v)$) gewählt werden. Die vorigen Überlegungen gelten dann analog.

Laufzeit des Entfernens

Theorem 4.3 (Laufzeit des Entfernens)**.** *Sei T ein binärer Suchbaum der Höhe h. Das Entfernen eines Schlüssels in T erfordert im schlechtesten Fall Zeit $\mathcal{O}(h)$.*

> **Beweis.** Der zu entfernende Knoten muss zunächst gefunden werden, was in Zeit $\mathcal{O}(h)$ möglich ist. Danach tritt einer der Fälle 1 bis 3 auf. In den ersten beiden Fällen werden lediglich Zeiger verändert und Speicher freigegeben. Daher fällt für sie nur Zeit $\mathcal{O}(1)$ an. Im dritten Fall muss der zunächst mithilfe des Algorithmus SymmetricSuccessor der symmetrische Nachfolger gefunden werden. Dies kostet maximal Zeit $\mathcal{O}(h)$, da die Pfadlänge von der Wurzel bis zu einem Blatt höchstens h beträgt. Danach werden lediglich die Schlüssel vertauscht und der symmetrische Nachfolger gelöscht, was in Zeit $\mathcal{O}(1)$ möglich ist. Damit wird im dritten Fall maximal Zeit $\mathcal{O}(h)$ benötigt. ∎

Durchlaufordnungen für Bäume Sei T ein binärer Suchbaum mit der Wurzel v, dem linken Teilbaum $T_l(v)$ und dem rechten Teilbaum $T_r(v)$. Wir können die Knoten von T auf verschiedene Arten durchlaufen:

Hauptreihenfolge Preorder

- In der *Hauptreihenfolge* (engl. *Preorder*) wird zunächst der in v gespeicherte Schlüssel ausgegeben. Danach wird zuerst rekursiv mit $T_l(v)$ fortgefahren und anschliessend rekursiv $T_r(v)$ verarbeitet.

Nebenreihenfolge Postorder

- In der *Nebenreihenfolge* (engl. *Postorder*) wird zunächst $T_l(v)$ rekursiv verarbeitet und danach $T_r(v)$. Schliesslich wird der in v gespeicherte Schlüssel ausgegeben.

Symmetrische Reihenfolge Inorder

- In der *symmetrischen Reihenfolge* (engl. *Inorder*) wird zunächst $T_l(v)$ rekursiv besucht, danach der in v gespeicherte Schlüssel ausgegeben und schliesslich $T_r(v)$ rekursiv besucht. Da alle Schlüssel in $T_l(v)$ kleiner und alle Schlüssel in $T_r(v)$ grösser sind als der in v gespeicherte Schlüssel, gibt die symmetrische Reihenfolge die in T gespeicherten Schlüssel in sortierter Reihenfolge aus.

Beispiel Die Knoten des binären Suchbaums aus Abbildung 4.13 werden in den folgenden Reihenfolgen durchlaufen:

- $7, 5, 10, 8, 9, 11, 15$ bei Verwendung der Hauptreihenfolge,
- $5, 9, 8, 15, 11, 10, 7$ bei Verwendung der Nebenreihenfolge,
- $5, 7, 8, 9, 10, 11, 15$ bei Verwendung der symmetrischen Reihenfolge.

Zusammenfassung und Ausblick Sei T ein binärer Suchbaum mit Höhe h, der n Schlüssel verwaltet. Binäre Suchbäume implementieren die Wörterbuchoperationen in Zeit $\mathcal{O}(h)$. Ausserdem werden eine Reihe weiterer Operationen unterstützt, zum Beispiel:

- MIN(T): Liefert den Schlüssel aus T mit minimalem Wert zurück. Diese Operation kann in Zeit $\mathcal{O}(h)$ realisiert werden, indem ausgehend von der Wurzel von T so lange dem linken Nachfolgerknoten gefolgt wird, bis dieser ein Blatt als linken Nachfolger besitzt. — MINIMALER SCHLÜSSEL
- EXTRACT-MIN(T): Sucht und entfernt den Schlüssel mit minimalem Wert aus T. Auch diese Operation kann in Zeit $\mathcal{O}(h)$ durchgeführt werden. — MINIMUM EXTRAHIEREN
- LIST(T): Liefert eine sortierte Liste der in T gespeicherten Schlüssel zurück. Diese Funktion wird von einem Durchlauf in symmetrischer Reihenfolge in Zeit $\mathcal{O}(n)$ realisiert. — SCHLÜSSEL AUSGEBEN
- JOIN(T_1, T_2): Seien T_1 und T_2 zwei Suchbäume zu den disjunkten Schlüsselmengen $\mathcal{K}_1$ und $\mathcal{K}_2$, und zusätzlich der maximale Wert eines Schlüssels in $\mathcal{K}_1$ kleiner als der minimale Wert eines Schlüssels in $\mathcal{K}_2$. JOIN(T_1, T_2) berechnet einen binären Suchbaum zur Schlüsselmenge $\mathcal{K}_1 \cup \mathcal{K}_2$. Dazu führen wir zunächst EXTRACT-MIN(T_2) aus, erhalten einen Schlüssel k sowie den aus T_2 resultierenden Baum T_2'. Dann erzeugen wir einen neuen Baum, dessen Wurzel den Schlüssel k speichert und die Teilbäume T_1 sowie T_2' besitzt. Die Laufzeit beträgt $\mathcal{O}(h)$. — VEREINIGUNG

Die soeben diskutierten natürlichen binären Suchbäume sind problematisch, wenn die Schlüssel (grösstenteils) in geordneter Reihenfolge eingefügt werden. Im schlechtesten Fall degenerieren sie zu einer linearen Liste, und die Laufzeiten für die Wörterbuchoperationen sind dann linear in der Anzahl der gespeicherten Schlüssel. — VERHALTEN IM SCHLECHTESTEN FALL

4.5 AVL-Bäume

Da die Laufzeit der Wörterbuchoperationen linear in der Baumhöhe sein kann, wäre es gut, wenn wir sicherstellen könnten, dass die Höhe stets sublinear oder besser noch in $\mathcal{O}(\log n)$ ist (wobei n wie zuvor die Anzahl der verwalteten Schlüssel bezeichnet). Wir haben bereits Baumstrukturen mit logarithmischer Höhe kennengelernt, nämlich Heaps. Leider sind diese viel zu unflexibel um dort dynamisch Schlüssel einzufügen und zu löschen, denn pro Anzahl verwalteter Schlüssel gibt es nur genau eine mögliche Struktur. Adelson-Velskii und Landis schlugen nun 1962 vor, dass sich in jedem Knoten eines binären Suchbaums die Höhen des links und des rechts gespeicherten Teilbaums um maximal 1 unterscheiden sollten (siehe Abbildung 4.17).

Sei nun T ein Baum mit der Wurzel v, der linke Teilbaum von v sei $T_l(v)$, und $T_r(v)$ der rechte (man beachte, dass sowohl $T_l(v)$ als auch $T_r(v)$ ein Blatt, d.h. ein Nullzeiger, sein können). Wir definieren die *Balance* des Knotens v als — STRUKTURBEDINGUNG

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v)), \tag{104}$$

wobei $h(T_l(v))$ bzw. $h(T_r(v))$ die Höhe von $T_l(v)$ bzw. $T_r(v)$ angeben. Die *AVL-Bedingung* besagt nun, dass für *alle* Knoten v des Baums $\text{bal}(v) \in \{-1, 0, 1\}$ gilt. Abbildung 4.18 zeigt Beispiele für AVL-Bäume. Um zu zeigen, dass die AVL-Bedingung — AVL-BEDINGUNG

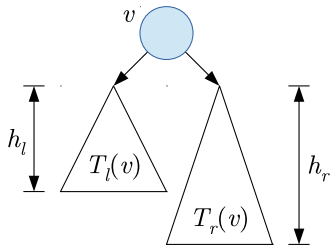


Abb. 4.17 Ein binärer Suchbaum mit der Wurzel v. Damit die AVL-Bedingung erfüllt ist, dürfen sich die Höhen h_l und h_r des linken bzw. des rechten Teilbaums von v maximal um 1 unterscheiden. Zudem muss diese Bedingung auch für alle Knoten gelten, die sich in $T_l(v)$ oder $T_r(v)$ befinden.

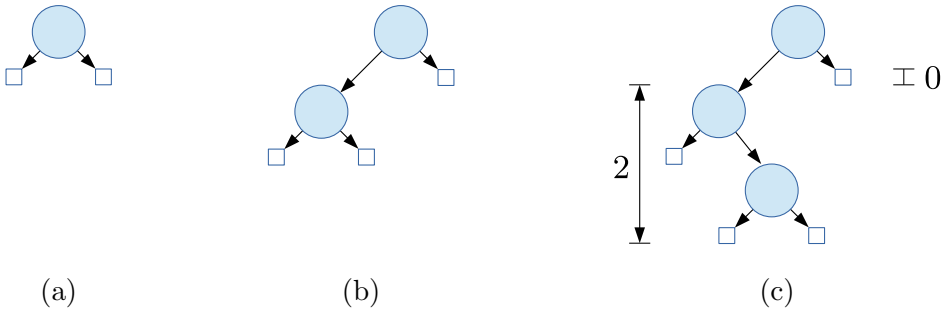


Abb. 4.18 Da sich bei AVL-Bäumen die Höhen der Teilbäume um maximal 1 unterscheiden dürfen, sind (a) und (b) Beispiele für AVL-Bäume, (c) hingegen nicht.

tatsächlich sinnvoll ist, zeigen wir zunächst, dass sie tatsächlich eine logarithmische Baumhöhe garantiert. Danach zeigen wir, wie in einen bestehenden AVL-Baum eingefügt werden kann, ohne die AVL-Eigenschaft zu verletzen.

Logarithmische Höhe Wir wollen für einen AVL-Baum mit n Schlüsseln zeigen, dass seine Höhe stets durch $\mathcal{O}(\log n)$ nach oben beschränkt ist. Leider ist ein direkter Nachweis dieser Aussage aber nicht so einfach, da es sehr viele mögliche AVL-Baumstrukturen zur Verwaltung von n Schlüsseln gibt. Daher bedienen wir uns eines

TRICK

Tricks: Statt für eine gegebene Schlüsselanzahl n zu untersuchen, welche Höhe ein AVL-Baum höchstens hat, untersuchen wir, wie viele Blätter ein AVL-Baum für eine gegebene Höhe h mindestens hat. Man kann leicht durch vollständige Induktion nachweisen, dass jeder binäre Suchbaum (und damit auch jeder AVL-Baum), der n Schlüssel speichert, genau $n + 1$ Blätter hat. Deswegen liefert eine solche untere Schranke für die Anzahl der Blätter in einem AVL-Baum mit einer gegebenen Höhe h automatisch eine obere Schranke für die Höhe eines AVL-Baums mit einer gegebenen Anzahl von Schlüsseln n.

MINDEST-BLATTZAHL

Verankerung I (AVL-Bäume mit Höhe $h = 1$): Es gibt nur einen AVL-Baum mit Höhe 1, nämlich denjenigen, der genau einen Schlüssel speichert und zwei Blätter besitzt (siehe Abbildung 4.18(a)). Die Mindestblattzahl eines AVL-Baums mit Höhe 1 beträgt daher $MB(1) = 2$.

Verankerung II (AVL-Bäume mit Höhe $h = 2$): Es gibt genau drei AVL-Bäume mit Höhe 2. Zwei davon verwalten genau zwei Schlüssel, von denen einer in der Wurzel des Baums und der andere im linken bzw. im rechten Nachfolger der

Wurzel gespeichert ist (siehe Abbildung 4.18(b)). Ausserdem gibt es noch einen Baum mit einer Wurzel und jeweils genau einem linken und einem rechten Nachfolger. Da jeder dieser Bäume mindestens drei Blätter hat, beträgt die Mindestblattzahl eines AVL-Baums mit Höhe 2 genau $MB(2) = 3$.

Rekursion (AVL-Bäume mit Höhe $h \geq 3$): Betrachte einen beliebigen AVL-Baum mit Höhe h. Dieser besteht aus einem Wurzelknoten v mit einem linken Teilbaum $T_l(v)$ und einem rechten Teilbaum $T_r(v)$ (siehe Abbildung 4.17). Die Höhe dieser Teilbäume ist höchstens $h-1$. Mindestens einer der Teilbäume muss Höhe $h-1$ haben (hätten beide eine Höhe strikt kleiner als $h-1$, dann wäre die Höhe des gesamten Baums strikt kleiner als h). Der andere Teilbaum hat entweder Höhe $h-1$, oder Höhe $h-2$, denn gemäss AVL-Bedingung dürfen sich die Höhen der Teilbäume höchstens um 1 unterscheiden. Da wir die *Mindestblattzahl* untersuchen, nehmen wir an, der andere Teilbaum hätte Höhe $h-2$ (denn ein Baum mit Höhe $h-1$ hat mit Sicherheit nicht weniger Blätter als ein Baum mit Höhe $h-2$). Die Anzahl der Blätter des gesamten Baums ist nun die Summe der Blattanzahlen in den beiden Teilbäumen. Es gilt also $MB(h) = MB(h-1) + MB(h-2)$.

Die Rekursionsformel für die Mindestblattzahl erinnert an die der Fibonacci-Zahlen. Diese waren als

$$F_1 := 1, \quad F_2 := 1, \quad F_h := F_{h-1} + F_{h-2} \text{ für } h \geq 3 \tag{105}$$

definiert. Damit gilt offenbar die Beziehung $MB(h) = F_{h+2}$, und man kann zeigen, dass $MB(h) \in \Theta\big((\frac{1+\sqrt{5}}{2})^h\big) \subseteq \Omega(1.6^h)$ gilt. Ein AVL-Baum mit Höhe h hat also mindestens 1.6^h viele Blätter, d.h., mindestens $1.6^h - 1$ viele Schlüssel. Im Umkehrschluss bedeutet dies, dass die Höhe eines AVL-Baums mit n Schlüsseln durch $1.44 \log_2 n$ nach oben beschränkt ist, also wie gehofft tatsächlich nur logarithmisch in der Anzahl der Schlüssel n ist. Das heisst also, ein AVL-Baum ist nur höchstens 44% höher als ein perfekt balancierter Binärbaum!

Einfügen in einen AVL-Baum Ein Schlüssel wird in einen AVL-Baum zunächst wie in einen natürlichen Suchbaum eingefügt. Danach testen wir, ob die AVL-Bedingung noch immer an allen Knoten gilt, und rebalancieren ansonsten den Baum. Wir beobachten zunächst, dass die AVL-Bedingung höchstens bei den Knoten auf dem Weg von der Wurzel zum neu eingefügten Schlüssel verletzt sein kann. Bei allen anderen Knoten ist die Balance unverändert, folglich gilt die AVL-Bedingung nach wie vor. Wir laufen also vom neu eingefügten Knoten zur Wurzel hoch und prüfen, ob die AVL-Bedingung noch immer gilt.

SPEICHERUNG DER BALANCE

Um effizient prüfen zu können, ob die AVL-Bedingung für einen Knoten erfüllt ist, könnten wir pro Knoten die Höhe des dort repräsentierten Teilbaums speichern. Das würde aber zu viel Platz brauchen, und es ausreichend, einfach nur die aktuelle Balance zu speichern, also -1 (linker Teilbaum höher), 0 (beide Teilbäume gleich hoch) oder $+1$ (rechter Teilbaum höher). Da man weiss, in welchen Teilbaum eingefügt wurde, weiss man auch, welcher Teilbaum ggf. gewachsen ist.

BALANCE PRÜFEN

Wir nehmen im Folgenden an, dass der neue Schlüssel in einen Knoten r eingefügt wurde, der der linke Nachfolger eines Knotens u ist. Der Fall, in dem r der rechte Nachfolger von u ist, kann völlig analog abgehandelt werden. Nun betrachten wir die bisherige Balance von u und unterscheiden drei (bzw. eigentlich nur zwei) Fälle.

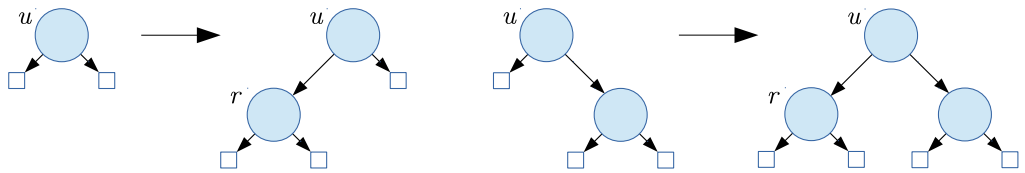


Abb. 4.19 Die Situation, in der u vor Einfügung des neuen Schlüssels in r (a) Balance 0, und (b) Balance 1 hatte.

1. Fall: bal$(u) = -1$. Solch ein Fall kann nicht auftreten, da u dann vor Einfügung des Schlüssels genau einen linken Nachfolger und keinen rechten Nachfolger gehabt hätte. Dann aber hätte r nicht linker Nachfolger von u werden können.

2. Fall: bal$(u) = 0$. Da r der linke Nachfolger von u ist, kann u vor der Einfügung des neuen Schlüssels keinen linken Nachfolger gehabt haben. Da die Balance von u vorher 0 war, kann u folglich auch keinen rechten Nachfolger gehabt haben. Der neue Knoten r ist der linke Nachfolger von u, also setzen wir bal$(u) \leftarrow -1$ und bal$(r) \leftarrow 0$ (siehe Abbildung 4.19(a)). Nun ist aber die Höhe des Teilbaums mit der Wurzel u um 1 gewachsen. Daher muss für die Vorgänger von r geprüft werden, ob die AVL-Bedingung noch gilt. Dazu rufen wir eine Methode UPIN(u) auf, deren Funktionsweise gleich erläutert wird.

3. Fall: bal$(u) = 1$. In diesem Fall hatte u vorher genau einen rechten Nachfolger. Da v der linke Nachfolger von u ist, setzen wir bal$(u) \leftarrow 0$ und bal$(r) \leftarrow 0$ (siehe Abbildung 4.19(b)). In diesem Fall ist die Höhe des Teilbaums mit der Wurzel u nicht gewachsen, also muss die AVL-Bedingung für alle Knoten des Baums nach wie vor gelten, und wir sind fertig.

Wir müssen uns also nur noch um den Fall kümmern, dass die AVL-Bedingung für einen Knoten auf dem Weg von u zur Wurzel des Baums verletzt ist. Dazu rufen wir, wie soeben erwähnt, eine Methode UPIN(u) auf. Diese Methode wird nur aufgerufen, wenn die folgenden drei Invarianten gelten:

INVARIANTEN FÜR UPIN

1) Der neu eingefügte Schlüssel befindet sich im Teilbaum mit der Wurzel u, und durch die Einfügung ist die Höhe des Teilbaums mit der Wurzel u um 1 gewachsen,

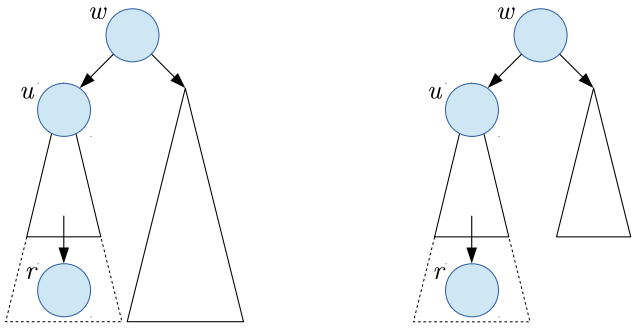
2) u hat eine von Null verschiedene Balance, und

3) u hat einen Vorgänger w.

BESCHREIBUNG VON UPIN

Man beachte, dass die Balance von u bereits im vorigen Schritt korrekt berechnet wurde, die Balance des Vorgängers w von u aber noch aktualisiert werden muss. Dazu nehmen wir o.B.d.A. an, dass u der linke Nachfolger von w ist (der Fall, dass u der rechte Nachfolger von w ist, wird analog abgehandelt), und unterscheiden wie zuvor drei Fälle:

1. Fall: bal$(w) = 1$. Dann war bisher der rechte Teilbaum von w um 1 höher als der linke Teilbaum mit der Wurzel u. Da dessen Höhe um 1 gewachsen ist, haben jetzt beide Teilbäume von w die gleiche Höhe, und wir setzen bal$(w) \leftarrow 0$ (siehe Abbildung 4.20(a)). Da sich die Höhe des Teilbaums mit der Wurzel w nicht verändert hat, sind wir fertig.



(a) (b)

Abb. 4.20 Situation beim Rebalancieren, wenn w vorher (a) Balance 1 hatte, und (b) Balance 0 hatte. Der gestrichelte Teil zeigt die Situation nach Einfügung des neuen Schlüssels in den Knoten r. Der Einfachheit halber wurden die Blätter von r nicht eingezeichnet.

2. Fall: bal$(w) = 0$. Dann hatten bisher beide Teilbäume von w die gleiche Höhe. Da wir in den Teilbaum mit der Wurzel u eingefügt haben und dessen Höhe um 1 gewachsen ist, setzen wir bal$(w) \leftarrow -1$ (siehe Abbildung 4.20(b)). Jetzt ist aber die Höhe des Teilbaums mit der Wurzel w um 1 gewachsen, also rufen wir UPIN(w) rekursiv auf (sofern auch w noch einen Vorgänger hat und nicht die Wurzel des gesamten Baums ist). Wir beobachten, dass der neue Schlüssel in den Teilbaum mit der Wurzel w eingefügt wurde und w eine von 0 verschiedene Balance hat, also sind die Invarianten für den Aufruf von UPIN(w) wieder erfüllt.

3. Fall: bal$(w) = -1$. Dann war der linke Teilbaum (mit der Wurzel u) bereits vor Einfügung des neuen Schlüssels um 1 höher als der rechte Teilbaum. Da die Höhe des linken Teilbaums durch die Einfügung des neuen Schlüssels erneut um 1 gewachsen ist, beträgt die Balance von w damit aktuell -2; damit ist die AVL-Bedingung bei w derzeit also verletzt. Zur Reparatur unterscheiden wir noch einmal zwei Fälle:

- Ist bal$(u) = -1$, dann wurde der neue Schlüssel in den linken Teilbaum von u eingefügt. Wir führen nun eine *Rechtsrotation* um w durch, bei der u die neue Wurzel des aktuellen Teilbaums wird, w rechter Nachfolger von u, und die Teilbäume von u und w entsprechend umgehängt werden (siehe Abbildung 4.21). Die Höhe des so erhaltenen Teilbaums mit der Wurzel u entspricht exakt der Höhe des früheren Teilbaums mit der Wurzel w vor der Einfügung des neuen Schlüssels, also sind wir fertig.

  EINFACHE ROTATION

- Ist dagegen bal$(u) = 1$, dann wurde der neue Schlüssel in den rechten Teilbaum von u eingefügt. Sei v der rechte Nachfolger von u. Wir nehmen o.B.d.A. an, dass der neue Schlüssel in den linken Teilbaum von v eingefügt wurde (der Fall, in dem der neue Schlüssel in den rechten Teilbaum von v eingefügt wurde, wird absolut identisch gehandhabt). Man beachte, dass v nicht den neu eingefügten Schlüssel enthält, denn dann hätte wäre der linke Nachfolger von w ein Blatt, und dieser Fall wurde bereits früher abgehandelt.

  Zur Wiederherstellung der AVL-Bedingung wird eine *Doppelrotation* erforderlich. Dazu führen wir zunächst eine einfache Linksrotation um u

  DOPPELROTATION

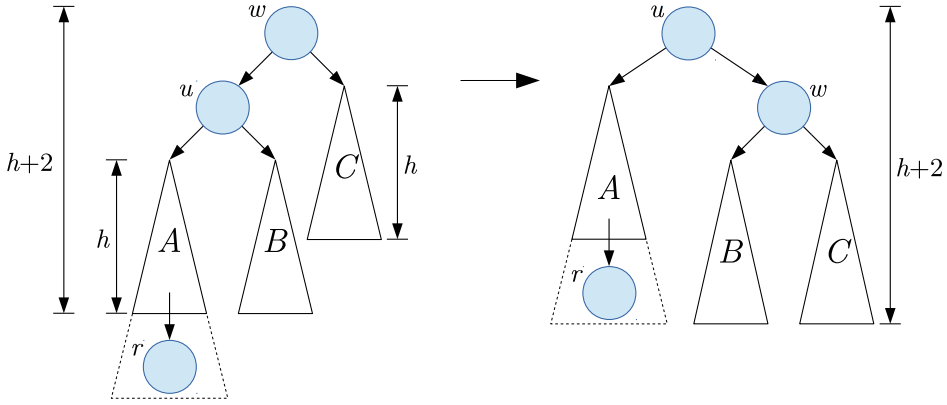


Abb. 4.21 Wurde der neue Schlüssel in den linken Teilbaum von u eingefügt, dann genügt eine einfache Rechtsrotation um den Knoten w zur Wiederherstellung der AVL-Bedingung bei w. Wie zuvor sind die Blätter von r nicht eingezeichnet.

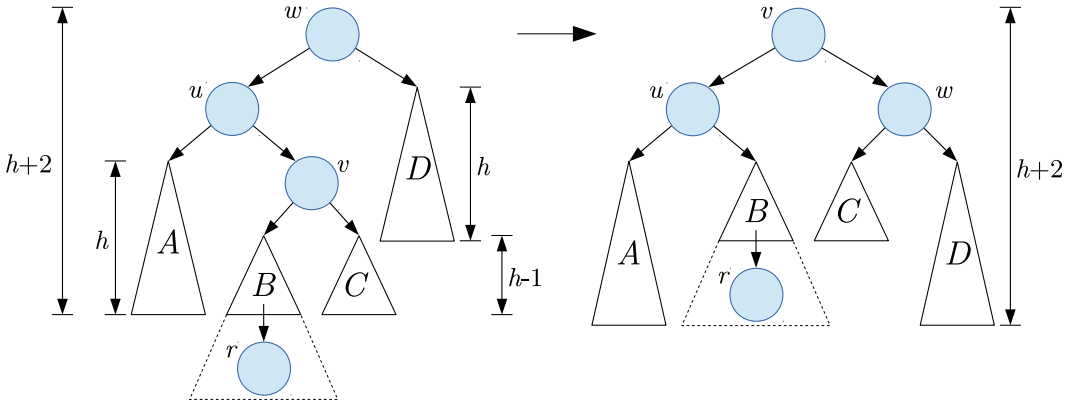


Abb. 4.22 Wurde der neue Schlüssel in den rechten Teilbaum von u eingefügt, dann wird eine Doppelrotation zur Rebalancierung benutzt. Dazu wird zunächst eine einfache Linksrotation um u und direkt danach eine einfache Rechtsrotation um w durchgeführt, sodass sich der rechts dargestellte Baum ergibt. Der Fall, in dem der neue Schlüssel in C (statt in B) eingefügt wurde, wird identisch gehandhabt.

durch, sodass u der linke Nachfolger von v und v der linke Nachfolger von w wird. Direkt danach führen wir eine einfache Rechtsrotation um w durch, sodass v die neue Wurzel des Teilbaums wird (siehe Abbildung 4.22).

Wie bei der einfachen Rotation hat der so erhaltene Teilbaum mit der Wurzel v die gleiche Höhe wie der frühere Teilbaum mit der Wurzel w vor der Einfügung des neuen Schlüssels, also sind wir fertig.

Wir sehen also, dass beim Einfügen nur eine einzige einfache oder Doppelrotation nötig ist, um die AVL-Bedingung bei allen Knoten wiederherzustellen.

LAUFZEIT

AVL-Bäume wurden eingeführt, damit die Wörterbuchoperationen bessere Laufzeiten im schlechtesten Fall haben. Um diese zu bestimmen, muss der Einfügevorgang und insbesondere die zum Rebalancieren benötigte Zeit genauer analysiert werden. Das Einfügen selbst (ohne Rebalancierung) hat eine in der Höhe des Baums beschränkte Laufzeit, und da die Höhe in $\mathcal{O}(\log n)$ liegt, folgt die gleiche obere Schran-

ke für die Laufzeit des Einfügens (ohne Rebalancierung). Die Methode UPIN führt (ohne den ggf. durchgeführten rekursiven Aufruf) nur konstant viele Operationen durch. Man beachte, dass auch Rotationen in konstanter Zeit durchgeführt werden kann, da bei ihnen nur konstant viele Vergleiche und Umhängungen von Zeigern anfallen. Bei jedem rekursiven Aufruf verringert sich der Abstand zur Wurzel um 1. Da die Höhe stets in $\mathcal{O}(\log n)$ liegt, gibt es also höchstens so viele rekursive Aufrufe, und da jeder nur konstante Zeit benötigt, läuft UPIN in Zeit $\mathcal{O}(\log n)$. Damit kann in einem AVL-Baum tatsächlich in Zeit $\mathcal{O}(\log n)$ gesucht und eingefügt werden.

LÖSCHEN

Die gleiche Laufzeit ergibt sich auch für das Löschen eines Schlüssels, das ähnlich zum Einfügen durchgeführt wird. Dort ist allerdings zu beachten, dass u.U. $\Theta(\log n)$ viele Rotationen zur Wiederherstellung der AVL-Bedingung benötigt werden.

Kapitel 5

Graphenalgorithmen

5.1 Grundlagen

Im historischen Königsberg (heute Kaliningrad) gab es sieben Brücken über den Fluss, den Pregel, die verschiedene Teile der Stadt miteinander verbanden. Abbildung 5.1 skizziert die Situation zu Beginn des 18. Jahrhunderts. Wir wollen nun überlegen, ob es einen Rundweg durch die Stadt gibt, die jede Brücke genau einmal überquert und am Ende zum Ausgangspunkt zurückkehrt.

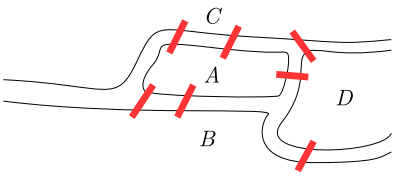


Abb. 5.1 Situation in Königsberg zu Beginn des 18. Jahrhunderts. Rot eingezeichnet sind die Brücken, die den Fluss (den Pregel) überqueren.

Wie immer abstrahieren wir zunächst. Wir erzeugen für jeden der vier Teile der Stadt einen *Knoten* und verbinden zwei Knoten durch eine *Kante*, wenn eine Brücke zwischen den entsprechenden Teilen besteht. Je mehr Brücken zwischen zwei Teilen existieren, umso mehr Kanten zwischen den entsprechen Knoten erzeugen wir. Eine solche Struktur nennt man einen *Graphen*. Abbildung 5.2 zeigt einen Graphen, der die in Abbildung 5.1 dargestellte Situation modelliert. Wir wollen jetzt untersuchen, ob es einen Rundweg (einen sog. *Zyklus*) durch den Graphen gibt, der jede Kante *genau einmal* benutzt. Ein solcher Zyklus wird *Eulerscher Zyklus* genannt (zu Ehren von Leonhard Euler, der 1736 nachwies, dass ein solcher Rundweg in Königsberg nicht existiert).

Knoten
Kante
Graph
Zyklus
Eulerscher Zyklus

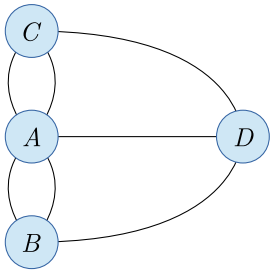


Abb. 5.2 Ein Graph, der die Situation aus Abbildung 5.1 modelliert. Die blauen Kreise stellen die Knoten da, die Linien die Kanten.

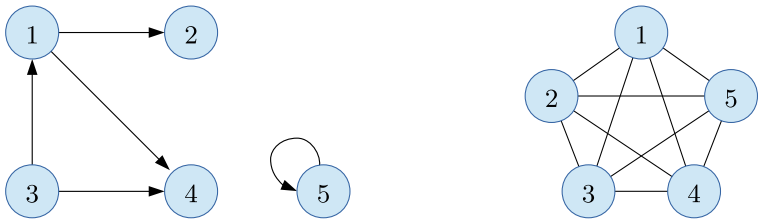


Abb. 5.3 Beispiele für Graphen: Gerichteter Graph mit fünf Knoten und einer Schleife um den Knoten 5 (a), ungerichteter vollständiger Graph mit fünf Knoten (b).

Wir beobachten nun: Falls ein Graph einen Eulerzyklus hat, dann gibt es an jedem Knoten eine gerade Anzahl Kanten (wir sagen dann auch, dass jeder Knoten einen geraden *Grad*) hat. Tatsächlich kann man sogar die Äquivalenz beweisen (was wir jetzt nicht tun): Ein Graph hat genau dann einen Eulerzyklus, wenn jeder Knoten einen geraden Grad hat. Bevor wir Beispiele für andere Probleme anschauen, definieren wir zunächst Graphen und die zugehörigen Konzepte formal.

5.1.1 Graphentheorische Grundlagen und Begriffe

Gerichteter Graph

Definition 5.1 (Gerichteter Graph)**.** *Ein* gerichteter Graph *besteht aus einer Menge* $V = \{v_1, \ldots, v_n\}$ *von* Knoten *(engl.* vertices*) und einer Menge* $E \subseteq V \times V$ *von* Kanten *(engl.* edges*).*

Ungerichteter Graph

Definition 5.2 (Ungerichteter Graph)**.** *Ein* ungerichteter Graph *besteht aus einer Menge* $V = \{v_1, \ldots, v_n\}$ *von* Knoten *und einer Menge* $E \subseteq \{\{u, v\} \mid u, v \in V\}$ *von* Kanten*.*

In gerichteten Graphen hat also jede Kante (u, v) eine Richtung; sie verläuft von u nach v. Eine Kante ist formal ein Paar von Knoten. In ungerichteten Graphen hat eine Kante keine Richtung. Sie ist formal eine Menge mit höchstens zwei Elementen. Kanten (v, v) (im Falle gerichteter Graphen) bzw. $\{v\}$ (im Falle ungerichteter Graphen) heissen *Schleifen*. Manchmal erlaubt man, dass E eine Multimenge ist, gleiche Kanten also mehrfach enthalten darf (dies ist z.B. bei dem in Abbildung 5.2 dargestellten Graph der Fall). Man spricht dann von einem *Multigraphen*. Für diese Vorlesung nehmen wir jedoch an, dass jeder Graph jede mögliche Kante höchstens einmal enthält, E also niemals eine Multimenge ist.

Schleife

Multigraph

Vollständiger Graph

Bipartiter Graph

Gewichteter Graph

Ein Graph $G = (V, E)$, in dem E jede Kante zwischen zwei verschiedenen Knoten v und w enthält, heisst *vollständig* (siehe Abbildung 5.3(b)). Ein Graph $G = (V, E)$, dessen Knotenmenge V in zwei disjunkte Mengen U und W aufgeteilt werden, sodass alle Kanten genau einen Knoten in U und einen Knoten und W haben, heisst *bipartit*. Ein *gewichteter Graph* $G = (V, E, c)$ ist ein Graph $G = (V, E)$ mit einer *Kantengewichtsfunktion* $c : E \to \mathbb{R}$. Der Wert $c(e)$ heisst dann *Gewicht* der Kante e, und ist oftmals nicht-negativ.

Adjazenz

Vorgänger

Nachfolger

Eingangsgrad

Ein Knoten w heisst *adjazent* zu einem Knoten v, falls E die Kante (v, w) (im Falle gerichteter Graphen) bzw. $\{v, w\}$ (im Falle ungerichteter Graphen) enthält. Für gerichtete Graphen $G = (V, E)$ ist die *Vorgängermenge* eines Knotens v als $N^-(v) := \{u \in V \mid (u, v) \in E\}$ definiert. Analog ist $N^+(v) := \{w \in V \mid (v, w) \in E\}$ die *Nachfolgermenge* eines Knotens v. Der *Eingangsgrad* eines Knotens v ist die Kardinalität der Vorgängermenge und wird mit $\deg^-(v)$ bezeichnet. Analog ist

(a) (b)

Abb. 5.4 Im gerichteten Graphen links (a) sind $\deg^-(v) = 3$, $\deg^+(v) = 2$, $\deg^-(w) = 1$ und $\deg^+(w) = 1$. Im ungerichteten Graphen rechts (b) sind $\deg(v) = 5$ und $\deg(w) = 2$.

der *Ausgangsgrad* eines Knotens v die Kardinalität der Nachfolgermenge und wird mit $\deg^+(v)$ bezeichnet. Das Analogon für ungerichtete Graphen ist die Menge $N(v) := \{w \in V \mid \{v, w\} \in E\}$, die als *Nachbarschaft* von v bezeichnet wird. Da in ungerichteten Graphen der Eingangs- und der Ausgangsgrad eines jeden Knotens v übereinstimmen, spricht man hier einfach nur vom *Grad* von v, und bezeichnet ihn mit $\deg(v)$. Hier muss allerdings ein Spezialfall beachtet werden: Eine Schleife um einen Knoten v erhöht den Grad in *ungerichteten Graphen* um 2 (und nicht um 1) erhöht; in gerichteten Graphen dagegen werden Schleifen nur einfach gezählt und erhöhen den Eingangs- und Ausgangsgrad eines Knotens lediglich um 1. Zwischen den Knotengraden und der Kantenanzahl bestehen nun die folgenden Beziehungen:

AUSGANGSGRAD

NACHBARSCHAFT

GRAD

Lemma 5.3. *In jedem Graphen $G = (V, E)$ gilt*

(i) $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$*, falls G gerichtet ist, und*

(ii) $\sum_{v \in V} \deg(v) = 2|E|$*, falls G ungerichtet ist.*

Eine Sequenz von Knoten $\langle v_1, \ldots, v_{k+1} \rangle$ heisst *Weg*, wenn für jedes $i \in \{1, \ldots, k\}$ eine Kante von v_i nach v_{i+1} existiert. Die Länge eines Weges ist k, die Anzahl der enthaltenen Kanten. In einem gewichteten Graphen ist das *Gewicht* eines Weges definiert als die Summe der Gewichte aller im Weg enthaltenen Kanten (v_i, v_{i+1}) (bzw. $\{v_i, v_{i+1}\}$ bei ungerichteten Graphen). Ein Weg, der keinen Knoten mehrfach benutzt, heisst *Pfad*. Ein Pfad, der in einem Knoten s startet und in einem Knoten t endet, heisst *st-Pfad*. Ein ungerichteter Graph $G = (V, E)$, in dem zwischen *jedem* Paar zweier Knoten v und w ein Weg existiert, heisst *zusammenhängend*.

WEG

PFAD

ZUSAMMENHANG

Ein Weg $\langle v_1, \ldots, v_k, v_{k+1} \rangle$ mit $v_1 = v_{k+1}$ heisst *Zyklus*. In einem Zyklus stimmen also Start- und Endknoten überein. Ein Zyklus heisst *Kreis*, falls er kein Knoten (mit Ausnahme des ersten und des letzten Knotens) und keine Kante mehr als einmal benutzt. Insbesondere ist eine Schleife ein Kreis der Länge 1. Man beachte, dass ungerichtete Graphen mit dieser Definition niemals Kreise der Länge 2 haben können. Ein Graph ohne jegliche Kreise heisst *kreisfrei*.

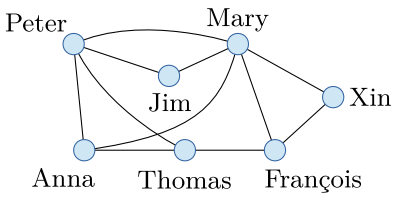
ZYKLUS

KREIS

KREISFREIER GRAPH

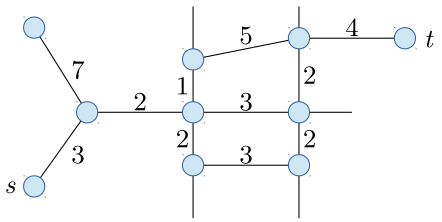
5.1.2 Beispiele für Graphen und Graphprobleme

Soziale Netzwerke Soziale Netzwerke wie etwa Facebook können durch Graphen beschrieben werden, in denen ein Knoten pro Account erzeugt wird und zwei Knoten miteinander verbunden werden, wenn die entsprechenden Accounts miteinander befreundet sind. Sind Freundschaften symmetrisch (wie z.B. bei Facebook), dann ist der Graph ungerichtet, ansonsten (wie z.B. bei Twitter) gerichtet. Für solche

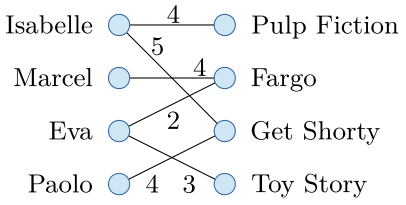


Graphen kann man zum Beispiel fragen, wie weit (also über wie viele Klicks) Xin von Peter entfernt ist. Das entsprechende graphentheoretische Problem besteht dann in der Berechnung der Länge eines *kürzesten Weges* von einem gegebenen Startknoten s zu einem gegebenen Zielknoten t.

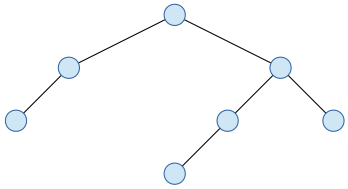
Strassennetze Strassennetze können durch gewichtete Graphen modelliert werden, indem man für jede Kreuzung einen Knoten erzeugt und zwei Knoten miteinander verbunden werden, wenn es zwischen den Kreuzungen eine direkte Verbindung (d.h. ohne weitere Zwischenkreuzungen) gibt. Das Gewicht einer Kante gibt dann z.B. an, wie lange die Traversierung der entsprechenden Kante durchschnittlich benötigt. Oftmals werden gerichtete Graphen benutzt, damit auch Einbahnstrassen (und manchmal sogar eigene Spuren) modelliert werden können. Das graphentheoretische Problem besteht nun darin, für zwei gegebene Knoten s und t einen Weg zu berechnen, dessen Kantenkosten minimal unter allen Wegen von s nach t ist.



Bewertungen Bei Streamingdiensten wie Netflix können Benutzer gesehene Filme bewerten. Auch solche Zusammenhänge können mit gewichteten Graphen modelliert werden, indem man zum Beispiel einen Knoten für jeden Benutzer und einen Knoten für jeden Film erzeugt. Der Graph enthält eine Kante für jede Wertung, die ein Zuschauer einem Film gibt, und das Gewicht dieser Kante spiegelt die numerische Bewertung wieder. Dieser Graph hat die Eigenschaft, dass er bipartit ist, denn Knotenmenge V kann so in zwei disjunkte Mengen U (hier: Benutzer) und W (hier: Filme) aufgeteilt werden, dass Kanten stets zwischen einem Knoten aus U und einem Knoten aus W verlaufen.



Bäume Auch Bäume können durch Graphen repräsentiert werden. Ein *Baum* ist ein ungerichteter Graph, der zusammenhängend und kreisfrei ist. Diese Definition weicht zunächst von der bisherigen Definition von Bäumen ab. Sobald man aber einen Knoten des hier definierten Baums als Wurzel festlegt und die Kanten von der Wurzel abwärts richtet, erhält man einen *gewurzelten* Baum, für den z.B. untersucht werden kann, ob er die AVL-Bedingung erfüllt. Auch kann man z.B. zeigen, dass jeder Baum mit n Knoten genau $n-1$ Kanten hat.



5.1.3 Datenstrukturen für Graphen

Adjazenzmatrix

Im Folgenden sei $G = (V, E)$ ein gerichteter Graph mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$. G kann durch seine sog. *Adjazenzmatrix* $A_G = (a_{ij})_{1 \leq i,j \leq n}$ repräsentiert werden, die die Grösse $n \times n$ hat und deren Einträge aus $\{0, 1\}$ stammen. Für $i, j \in \{1, \ldots, n\}$ ist der Eintrag a_{ij} genau dann 1, wenn E die Kante (v_i, v_j) enthält. Ungerichtete Graphen können natürlich analog repräsentiert werden: Enthält E eine

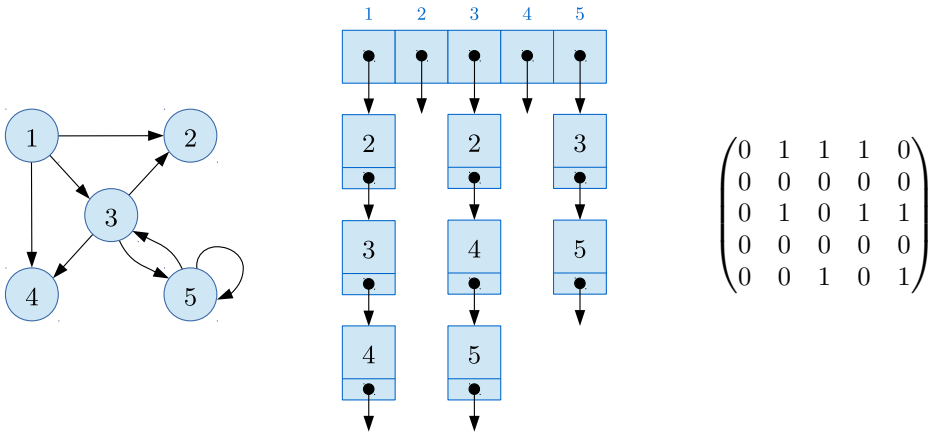


Abb. 5.5 Ein gerichteter Graph mit $n = 5$ Knoten (links), sowie seine Darstellung als Adjazenzliste (Mitte) und als Adjazenzmatrix (rechts). Die Knoten in den einzelnen Listen können, müssen aber nicht nach dem Index der Nachfolgerknoten aufsteigend sortiert sein.

Kante $\{v_i, v_j\}$, dann haben in der Adjazenzmatrix beide Einträge a_{ij} und a_{ji} den Wert 1, sonst 0. Hat ein Graph (gerichtet oder ungerichtet) eine Schleife um einen Knoten v_i, dann enthält der i-te Diagonaleintrag (also a_{ii}) der Adjazenzmatrix eine 1. Die Adjazenzmatrix benötigt offenbar Platz $\Theta(|V|^2)$, da sie $|V|^2$ Einträge hat und pro Eintrag genau ein Bit benötigt wird.

PLATZBEDARF

Wir haben bereits früher die Berechnung von kürzesten Wegen in Strassennetzen als Anwendung für Graphen genannt. Strassennetze haben aber die Eigenschaft, dass zu einer Kreuzung (Knoten) oftmals verhältnismässig wenige Strassen (Kanten) führen, meist vier oder weniger. Reale Strassennetze (etwa von Kontinentaleuropa) haben aber oft mehrere Millionen Knoten. Das bedeutet also: Wird ein Strassennetz durch eine Adjazenzmatrix repräsentiert, dann hat jede Zeile (die die Nachbarn des entsprechenden Knotens codiert) höchstens vier oder fünf Einsen, aber mehrere Millionen Nullen. Offenbar wird hier also sehr viel Platz verschenkt. Zum Glück gibt es eine weitere Art, Graphen zu repräsentieren, nämlich die sog. *Adjazenzlistendarstellung*. Eine Adjazenzliste besteht aus einem Array $(A[1], \ldots, A[n])$, wobei der Eintrag $A[i]$ eine einfach verkettete Liste aller Knoten in $N^+(v)$ (bzw. deren Indizes) enthält. Die Reihenfolge, in der die in $A[i]$ gespeicherte Liste die Nachfolger von v_i (bzw. deren Indizes) enthält, ist beliebig. Sie muss also insbesondere nicht nach Index aufsteigend sortiert sein. Die Adjazenzlistendarstellung benötigt Platz $\Theta(|V| + |E|)$, denn das Array besitzt $|V|$ Plätze, und die die Listen enthalten insgesamt $|E|$ Listenelemente. Ist der Graph nur dünn besetzt, zum Beispiel wenn $|E| \in \mathcal{O}(|V|)$ ist (was zum Beispiel für Graphen zur Modellierung von Strassennetzen erfüllt ist), dann benötigt die Adjazenzliste deutlich weniger Platz als die Adjazenzmatrix.

ADJAZENZLISTE

PLATZBEDARF

Adjazenzmatrizen und -listen unterstützen verschiedene Operationen unterschiedlich gut. Tabelle 5.6 zeigt eine Übersicht.

Alle Nachbarn von $v \in V$ ermitteln	$\Theta(n)$	$\Theta(\deg^+(v))$
Finde $v \in V$ ohne Nachbar	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Ist $(u, v) \in E$?	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante löschen	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$

Abb. 5.6 Operationen und ihre Laufzeiten auf Adjazenzmatrizen und -listen.

5.1.4 Beziehung zur Linearen Algebra

Betrachten wir noch einmal die Adjazenzmatrix des in Abbildung 5.5 dargestellten Graphen und quadrieren wir sie, dann erhalten wir die Matrix

$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix} \tag{106}$$

Wie kann diese anschaulich interpretiert werden? Ein Eintrag b_{ij} dieser Matrix wird durch $b_{ij} = \sum_{k=1}^{n} a_{ik} \cdot a_{kj}$ berechnet. Wir beobachten nun, dass das Produkt $a_{ik} \cdot a_{kj}$ entweder 0 oder 1 ist, und es ist genau dann 1, wenn sowohl $(v_i, v_k) \in E$ als auch $(v_k, v_j) \in E$ sind. Das heisst also, b_{ij} zählt die Anzahl der Wege der Länge 2 von i nach j. Wir können die Aussage sogar auf A_G^t für beliebige $t \in \mathbb{N}$ verallgemeinern, wie das folgende Theorem zeigt:

Anzahl Wege der Länge t

Theorem 5.4. *Sei G ein Graph (gerichtet oder ungerichtet) und $t \in \mathbb{N}$ beliebig. Das Element an der Position (i, j) in der Matrix A_G^t gibt die Anzahl der Wege der Länge t von v_i nach v_j an.*

Beweis. Wir beweisen die Aussage durch vollständige Induktion über t.

Induktionsanfang ($t = 1$): Die Adjazenzmatrix $A_G = A_G^1$ enthält die Anzahl der Wege der Länge 1. Zwischen zwei Knoten gibt es entweder einen Weg der Länge 1 (wenn die entsprechende Kante existiert), und keinen sonst.

Induktionshypothese: Angenommen, in A_G^t gibt jeder Eintrag (i, j) die Anzahl der Wege der Länge t von v_i nach v_j an.

Induktionsschritt ($t \to t + 1$): Sei $B = (b_{ij})_{1 \le i,j \le n} := A_G^t$. Betrachte nun $C = (c_{ij})_{1 \le i,j \le n} := A_G^{t+1} = B \times A_G$. Für einen Eintrag c_{ij} gilt

$$c_{ij} = \sum_{k=1}^{n} b_{ik} \cdot a_{kj}. \tag{107}$$

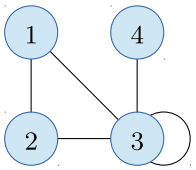
Dabei zählt b_{ik} gemäss Induktionshypothese die Anzahl der Wege von v_i nach v_k der Länge t. Ausserdem ist a_{kj} entweder 0 oder 1, und 1 genau dann, wenn eine Kante (also ein Weg der Länge 1) von v_k nach v_j existiert. Das heisst, die Anzahl der Wege von v_i nach v_j über den Zwischenknoten v_k ist genau b_{ik}, falls (v_k, v_j) existiert, und 0 sonst. Da die Summe über alle möglichen Zwischenknoten v_k gebildet wird, gibt c_{ij} also korrekt die Anzahl der Wege von v_i nach v_j der Länge $t + 1$ an. ∎

Anwendungen

Um in einem Graphen den kürzesten Weg (mit einer minimalen Anzahl von Kanten) zwischen zwei gegebenen Knoten v_i und v_j zu bestimmen, genügt es also offenbar, A_G zu potenzieren, bis zum ersten Mal der Eintrag an der Position (i, j) ungleich Null ist. Da ein *kürzester* Weg offenbar keinen Knoten mehrfach benutzt und damit höchstens Länge $n - 1$ hat, reicht eine Potenzierung bis n aus; enthält die Position (i, j) dann noch immer keine Eins, dann gibt es keinen Weg von v_i nach v_j.

Eine andere Anwendung besteht in der Berechnung der Anzahl von Dreiecken in ungerichteten Graphen. Ein *Dreieck* ist eine Menge D von drei Knoten, wobei zwischen jedem Paar zweier Knoten in D eine Kante verläuft. Um für einen gegebenen ungerichteten Graphen G die Anzahl der Dreiecke zu bestimmen, entfernen wir zunächst alle Schleifen von G, indem wir die Diagonaleinträge in A_G auf 0 setzen. Danach berechnen wir die Matrix $B = (b_{ij})_{1 \leq i,j \leq n} := A_G^3$ (dazu genügen zwei Matrixmultiplikationen). Ein Eintrag b_{ii} gibt nun an, wie viele Wege der Länge 3 von v_i nach v_i existieren, d.h., er zählt die Anzahl der Dreiecke in denen v_i enthalten ist. Da jedes Dreieck irgendeinen Knoten des Graphen enthält und auf sechs verschiedene Arten gezählt werden wird ($\langle x, y, z, x \rangle$, $\langle x, z, y, x \rangle$, $\langle y, z, x, y \rangle$, $\langle y, x, z, y \rangle$, $\langle z, x, y, z \rangle$, $\langle z, y, x, z \rangle$), beträgt die Anzahl der Dreiecke exakt $(\sum_{i=1}^{n} b_{ii})/6$. Als Beispiel betrachten wir den folgenden Graph:

Dreieck



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

Nach Entfernung der Schleifen ergeben sich die neue Adjazenzmatrix bzw. ihre dritte Potenz

$$\bar{A}_G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ bzw. } B = \bar{A}_G^3 = \begin{pmatrix} 2 & 3 & 4 & 1 \\ 2 & 2 & 4 & 1 \\ 4 & 4 & 2 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix}.$$

Offenbar hat G genau ein Dreieck (nämlich die Menge $\{v_1, v_2, v_3\}$), und die Summe der Diagonaleinträge in B ist genau 6.

5.1.5 Beziehung zu Relationen

Einen Graphen $G = (V, E)$ können wir auch als Relation $E \subseteq V \times V$ auf V auffassen, und umgekehrt. Sei wie zuvor $A_G = (a_{ij})_{1 \leq i,j \leq n}$ die Adjazenzmatrix von G. Konzepte von Relationen können wie folgt auf Graphen übertragen werden:

- E ist genau dann *reflexiv*, wenn E jede Kante (v, v) mit $v \in V$ enthält, G also eine Schleife um jeden Knoten $v \in V$ hat. Dies ist genau dann der Fall, wenn $a_{ii} = 1$ für alle $i \in \{1, \ldots, n\}$ ist. Reflexivität
- E ist genau dann *symmetrisch*, wenn für alle $(v, w) \in E$ auch $(w, v) \in E$ gilt, d.h. G ungerichtet ist. Symmetrie
- E ist genau dann transitiv, wenn für jedes Paar zweier Kanten $(u, v) \in E$ und $(v, w) \in E$ auch $(u, w) \in E$ ist. Transitivität

Eine Äquivalenzrelation erfüllt bekanntermassen alle drei obigen Eigenschaften. Folglich ist der Graph zu einer Äquivalenzrelation eine Kollektion vollständiger, ungerichteter Graphen, wobei jeder Knoten eine Schleife hat.

Die reflexive und transitive Hülle eines Graphens $G = (V, E)$ beschreibt die sog. *Erreichbarkeitsrelation* E^*. Dies ist eine zweistellige Relation, die ein Paar (v, w) genau dann enthält, wenn es in G einen Weg vom Knoten v zum Knoten w gibt. Erreichbarkeitsrelation

5.1.6 Berechnung der reflexiven und transitiven Hülle

Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V, E)$ durch seine Adjazenzmatrix $A_G = (a_{ij})_{1 \le i,j \le n}$. Wir wollen nun die reflexive und transitive Hülle von G als Matrix berechnen. Das Ziel ist also die Berechnung einer Matrix $B = (b_{ij})_{1 \le i,j \le n}$, wobei b_{ij} genau dann den Wert 1 annimmt, wenn $(v_i, v_j) \in E^*$ ist (und 0 sonst). Unser Algorithmus darf in-place arbeiten, d.h. er darf die Matrix A überschreiben, sodass am Ende an ihrer Stelle B steht. Wir beobachten, dass es niemals nötig ist, eine 1 zu überschreiben, da eine 1 an einer Position (i, j) signalisiert, dass es (irgendeinen) Weg von v_i nach v_j gibt. Nur Nullen müssen überschrieben werden, und zwar genau dann, wenn ein Weg zwischen zwei Knoten gefunden wird und bisher nicht bekannt war, dass zwischen ihnen ein Weg verläuft.

IN-PLACE

ERSTE IDEE

Um nun Wege der Länge 2 zu finden, können wir über alle $i, j, k \in \{1, \ldots, n\}$ iterieren und $a_{ij} \leftarrow 1$ setzen, wenn sowohl $a_{ik} = 1$ als auch $a_{kj} = 1$ sind. Zusätzlich setzen wir natürlich $a_{ii} \leftarrow 1$ für alle $i \in \{1, \ldots, n\}$, da die Hülle ja auch reflexiv sein soll. Nach diesen Schritten codiert die Matrix bereits alle Wege der Längen 1 und 2. Führen wir dieses Verfahren erneut aus, wurden alle Wege der Länge höchstens 4 gefunden, bei einer weiteren Ausführung alle Wege der Länge höchstens 8, usw. Daher reichen $\lceil \log_2 n \rceil$ viele Ausführungen, um alle Wege zu finden.

VERBESSERUNG

Tatsächlich reicht sogar eine einzige Ausführung, wenn wir sicherstellen können, dass sowohl a_{ik} als auch a_{kj} bereits ihren entgültigen Wert enthalten (also 1, falls es irgendeinen Weg von v_i nach v_k bzw. von v_k nach v_j gibt, und 0 sonst). Warshall beobachtete nun, dass es ausreicht, die Wege in der Reihenfolge des grössten Zwischenknotens v_k zu berechnen, was der folgende Algorithmus tut:

REFLEXIVE UND TRANSITIVE HÜLLE

ReflexiveTransitiveHull(A)

```
1 for k ← 1, ..., n do
2     a_kk ← 1                                  ▷ Reflexive Hülle
3     for i ← 1, ..., n do
4         for j ← 1, ..., n do
5             if a_ik = 1 and a_kj = 1 then a_ij ← 1    ▷ Berechne Weg über v_k
```

LAUFZEIT

Die Laufzeit dieses Algorithmus ist offenbar in $\Theta(n^3)$. Die Korrektheit weisen wir induktiv nach, dass in der äusseren Schleife die folgende Invariante stets erfüllt bleibt: Alle Wege, bei denen der grösste Zwischenknoten (und damit alle Zwischenknoten) einen Index $< k$ hat, wurden berücksichtigt.

Induktionsanfang ($k = 1$): Alle direkten Wege ohne Zwischenknoten (dies entspricht allen Kanten) sind bereits in A_G enthalten.

Induktionshypothese: Sei die obige Invariante vor dem k-ten Schleifendurchlauf wahr, d.h. alle Wege, bei denen der grösste Zwischenknoten einen Index $< k$ hat, wurden berücksichtigt.

Induktionsschritt ($k \to k + 1$): Betrachte nun den k-ten Schleifendurchlauf sowie einen beliebigen Weg von einem Knoten v_i zu einem Knoten v_j, dessen grösster Zwischenknoten den Index k hat. Wir müssen zeigen, dass der Algorithmus dann tatsächlich $a_{ij} \leftarrow 1$ setzt. Dies ist der Fall, denn nach Induktionshypothese haben sowohl a_{ik} als auch a_{kj} den Wert 1 (da k der grösste Index eines

Zwischenknotens auf einem Weg von v_i nach v_j ist, müssen die Zwischenknoten auf den Wegen von v_i nach v_k bzw. von v_k nach v_j entsprechend kleinere Indizes haben). Daher setzt der Algorithmus in Schritt 5 auch tatsächlich $a_{ij} \leftarrow 1$, und da auf diese Weise alle Wege gefunden werden, deren grösster Zwischenknoten den Index k hat, bleibt die Invariante erhalten. ∎

5.2 Durchlaufen von Graphen

Wir untersuchen jetzt, wie die Knoten eines Graphen in systematischer Art und Weise besucht werden können. Obwohl die Fragestellung vielleicht künstlich erscheinen mag, so hat sie dennoch viele Anwendungen, zum Beispiel, aus einem gegebenen Labyrinth herauszufinden; oftmals werden solche Knotendurchläufe auch als Teilroutine zum Erkundschaften der Struktur eines Graphen benutzt. Wir werden später auch noch sehen, wie einer beiden im Folgenden vorgestellten Algorithmen erweitert werden kann, um einen Weg mit minimalem Gesamtgewicht in einem gewichteten Graphen (der z.B. ein Strassennetz o.ä. modellieren könnte) zu berechnen. Für dieses und die folgenden Abschnitte nehmen wir an, dass der Graph als eine Adjazenzliste und nicht als Adjazenzmatrix gespeichert ist.

5.2.1 Tiefensuche

Rekursiver Algorithmus Die Idee bei der Tiefensuche ist es, bei einem Knoten zu starten und dann so lange einem dort startenden Pfad zu folgen, bis man nur noch bereits besuchte Knoten findet. Erst dann geht man einen Schritt (d.h., eine Kante) zurück und versucht dort, den Pfad zu erweitern. Ist dies ebenfalls nicht möglich, weil man alle möglichen Knoten ebenfalls bereits früher besucht hat, dann geht man einen weiteren Schritt zurück. Aus diesem Vorgehen resultiert auch der Name *Tiefensuche* (engl. *Depth First Search*, kurz *DFS*): Man verfolgt einen Pfad in die Tiefe, bis er nicht mehr erweitert werden kann.

IDEE

Wir markieren also stets den aktuell besuchten Knoten als besucht, iterieren über alle Nachfolger und setzen die Tiefensuche rekursiv bei genau den Nachfolgerknoten fort, die noch nie besucht wurden. Eine Tiefensuche in einem Graphen $G = (V, E)$, die bei einem Knoten $v \in V$ startet, kann wie folgt formuliert werden.

DFS-VISIT(G, v) TIEFENSUCHE

```
1 Markiere v als besucht
2 for each (v, w) ∈ E do
3     if w ist noch nicht besucht then
4         DFS-Visit(G, w)
```

Offenbar besucht dieser Algorithmus u.U. nicht alle Knoten des Graphen, nämlich genau dann, wenn nicht jeder Knoten von v aus erreichbar ist. Wenn so etwas passiert, dann starten wir die Suche einfach bei einem noch nie besuchten Knoten neu. Wir erhalten also folgendes Rahmenprogramm.

RAHMEN-PROGRAMM

DFS(G)

```
1 for each v ∈ V do
2     if v ist noch nicht besucht then
3         DFS-Visit(G, v)
```

Man beachte, dass die Reihenfolge, in der die Algorithmen die Knoten aus V (in DFS) bzw. die Nachfolger von v (in DFS-VISIT) nicht festgelegt ist. Üblicherweise wird die von der Adjazenzlistendarstellung implizit vorgegebene Reihenfolge benutzt. Dies bedeutet insbesondere, dass die Nachfolger eines Knotens genau dann in alphabetischer Reihenfolge abgearbeitet werden, wenn sie in der Adjazenzliste in alphabetischer Reihenfolge vorkommen, was aber nicht unbedingt der Fall sein muss.

LAUFZEIT

Laufzeit Um die Laufzeit der Tiefensuche zu bestimmen, beobachten wir zunächst, dass DFS-VISIT für jeden Knoten $v \in V$ genau einmal aufgerufen wird, denn jeder Knoten v wird beim ersten Aufruf von DFS-VISIT(G, v) als besucht markiert, und vor jedem Aufruf von DFS-VISIT (Schritt 3 in DFS-VISIT bzw. Schritt 2 im DFS-Rahmenprogramm) wird geprüft, ob der entsprechende Knoten schon früher besucht wurde. Also braucht DFS(G) *ohne* die Aufrufe von DFS-VISIT(G, v) Zeit $\Theta(n)$. In DFS-VISIT(G, v) fällt ohne die rekursiven Aufrufe Zeit $a \deg^+(v) + c$ (für geeignet gewählte Konstanten $a, c \in \mathbb{R}^+$) an, denn das Markieren kostet konstante Zeit und danach wird für alle Nachfolger w von v in konstanter Zeit geprüft, ob w schon früher besucht wurde. Damit beträgt die Gesamtlaufzeit einer Tiefensuche genau $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E| + |V|) = \Theta(|V| + |E|)$, was linear in der Eingabegrösse ist.

RAHMEN PROGRAMM

GESAMTLAUFZEIT

Beispiel Abbildung 5.7 zeigt das Ergebnis einer Tiefensuche auf einem Beispielgraphen. Für unser Beispiel nehmen wir an, dass alle Schleifen die entsprechenden Knoten stets in alphabetisch aufsteigender Reihenfolge betrachten. Der erste betrachtete Knoten ist damit a, und von dort ausgehend werden zunächst b, c und f entdeckt. Von f gibt es keine ausgehenden Kanten, also wird die Tiefensuche bei c fortgesetzt. Dort gibt es ebenfalls keine weiteren unentdeckten Nachbarn, weswegen die Tiefensuche zuerst bei b und dann bei a fortgesetzt wird. Von dort aus werden noch die Knoten d und e entdeckt, danach gibt es keine Knoten mehr, die von a aus erreichbar sind. Daher wird die Tiefensuche bei g neu gestartet (g ist der alphabetisch kleinste Knoten, der noch nicht besucht wurde), und von dort aus werden noch die Knoten h und i entdeckt.

Iterativer Algorithmus Die zuvor gezeigte rekursive Formulierung der Tiefensuche wird in einer praktischen Implementierung schnell an ihre Grenzen stossen, wenn sie auf einem sehr grossen Graphen (mit mehreren Millionen Knoten) eingesetzt werden soll. Wir können die Rekursion aber vermeiden, indem wir einen Stapel benutzen. In jedem Schritt nehmen wir den obersten Knoten v vom Stapel und gehen wie zuvor die Nachfolger von v durch (diesmal aber in umgekehrter Reihenfolge); wurde ein Nachfolger noch nie besucht, wird er auf den Stapel gelegt. Damit erhalten wir den folgenden Algorithmus.

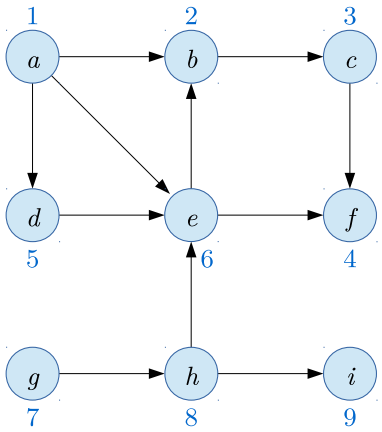


Abb. 5.7 Eine Tiefensuche auf dem dargestellten Graphen. Die blauen Zahlen neben jedem Knoten geben an, zu welchem Zeitpunkt ein Knoten erstmals entdeckt wird. Die Reihenfolge, in der die Knoten entdeckt werden, ist also $a, b, c, f, d, e, g, h, i$. Bei g findet ein Neustart statt, da dieser Knoten von a aus nicht erreicht werden kann.

Iterative Tiefensuche

DFS-Visit-Iterative(G, v)

```
1 S ← ∅                                   ▷ Initialisiere leeren Stapel S
2 Push(v, S)                              ▷ Leg v oben auf S
3 while S ≠ ∅ do
4     w ← Pop(S)                          ▷ Aktueller Knoten
5     if w noch nicht besucht then
6         Markiere w als besucht
7         for each (w, x) ∈ E in reverse order do
8             if x noch nicht besucht then
9                 Push(x, S)              ▷ Leg x oben auf S
```

Laufzeit

Man sieht leicht, dass die Laufzeit noch immer in $\Theta(|V|+|E|)$ liegt: Jeder Knoten w wird höchstens einmal als besucht markiert (genau dann, wenn er vom Startknoten v aus erreichbar ist), und nur dann, wenn er besucht wird, fällt ein Extraaufwand von $\Theta(\deg^+(w)+1)$ an. Der Stapel kann insgesamt bis zu $|E|$ viele Elemente enthalten, also ist die Gesamtlaufzeit in $\mathcal{O}(|V|+|E|+\sum_{w\in V}(\deg^+(w)+1)) = \mathcal{O}(|V|+|E|)$. Wird die Methode DFS-Visit-Iterative aus dem DFS-Rahmenprogramm heraus aufgerufen (anstelle von DFS-Visit), dann ist die Laufzeit insgesamt wieder in $\Theta(|V|+|E|)$.

Extraplatz

Ein Wermutstropfen der obigen iterativen Formulierung ist, dass im schlimmsten Fall tatsächlich Platz $\Theta(|E|)$ benötigt wird. Ein solcher Fall tritt zum Beispiel dann ein, wenn der Algorithmus auf einem vollständigen Graphen aufgerufen wird. Die rekursive Formulierung benötigt niemals mehr als $\Theta(|V|)$ viel Extraplatz, da die Rekursionstiefe durch $|V|$ nach oben beschränkt ist und pro rekursivem Aufruf nur konstant viel Extraplatz benötigt wird.

Verbesserung

Der obige Algorithmus kann aber so modifiziert werden, dass der Extraspeicher nur höchstens $\Theta(|V|)$ statt $\Theta(|E|)$ beträgt. Dazu legen wir nicht die entsprechenden Nachfolgerknoten auf den Speicher, sondern nur diejenigen Knoten, die bereits besucht aber noch nicht komplett abgearbeitet wurden, und speichern für jeden solchen Knoten auch einen Verweis auf den nächsten dort zu besuchenden Nachfolger (z.B. durch einen Zeiger auf den entsprechenden Knoten in der Adjazenzliste). Diese Modifikation ist nicht schwierig, aber recht technisch, weswegen sie im Folgenden nicht weiter diskutiert wird.

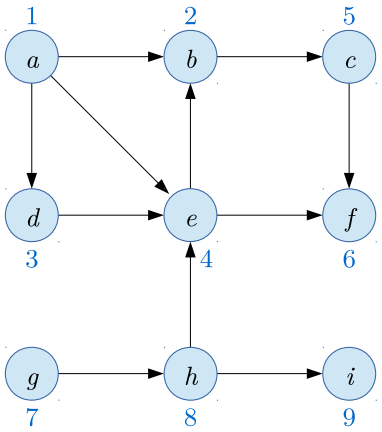


Abb. 5.8 Eine Breitensuche auf dem dargestellten Graphen. Die blauen Zahlen neben jedem Knoten geben an, zu welchem Zeitpunkt ein Knoten erstmals entdeckt wird. Die Reihenfolge, in der die Knoten entdeckt werden, ist also $a, b, d, e, c, f, g, h, i$. Bei g findet ein Neustart statt, da dieser Knoten von a aus nicht erreicht werden kann.

5.2.2 Breitensuche

Anstatt einen Pfad in die Tiefe zu verfolgen, könnten wir auch zuerst alle Nachfolger eines Knotens abarbeiten, dann alle Nachfolger dieser Nachfolger usw. Diese Idee wird von der *Breitensuche* aufgegriffen. Abbildung 5.8 zeigt noch einmal den zuvor gezeigten Graphen und zeigt auch, in welcher Reihenfolge eine Breitensuche die Knoten betrachtet, wenn (wie zuvor) die Nachfolger eines Knotens in alphabetisch aufsteigender Reihenfolge bearbeitet werden.

BREITENSUCHE

Algorithmisch können wir analog zur iterativen Tiefensuche vorgehen. Wir stellen jetzt aber direkt eine Variante vor, die mit $\mathcal{O}(|V|)$ Extraplatz auskommt. Dazu wird der Stapel durch eine Schlange ersetzt, und damit ein Knoten nicht mehrfach in die Schlange aufgenommen wird, markieren wir ihn als "aktiv", sobald er das erste Mal aufgenommen wurde. Ist ein Knoten abgearbeitet, wird er wie bisher als "besucht" markiert (die Markierung ist nicht unbedingt notwendig und dient nur der besseren Unterscheidbarkeit zu aktiven Knoten). Man überlege sich als Hausaufgabe, warum ein solcher einfacher Trick bei der iterativen Tiefensuche nicht funktioniert.

ITERATIVE BREITENSUCHE

BFS-VISIT-ITERATIVE(G, v)

```
1  Q ← ∅                                          ▷ Initialisiere leere Schlange Q
2  Markiere v als aktiv
3  ENQUEUE(v, Q)                                  ▷ Füge v zur Schlange Q hinzu
4  while Q ≠ ∅ do
5      w ← DEQUEUE(Q)                             ▷ Aktueller Knoten
6      Markiere w als besucht
7      for each (w, x) ∈ E do
8          if x nicht aktiv und x noch nicht besucht then
9              Markiere x als aktiv
10             ENQUEUE(x, Q)                      ▷ Füge x zur Schlange Q hinzu
```

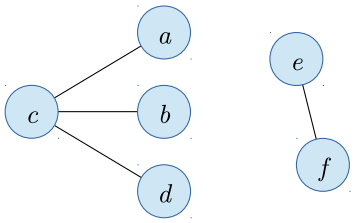


Abb. 5.9 Ein ungerichteter Graph mit zwei Zusammenhangskomponenten. Diese können durch die Knotenmengen $V_1 = \{a, b, c, d\}$ und $V_2 = \{e, f\}$ eindeutig beschrieben werden.

Als Rahmenprogramm für die Breitensuche kann das gleiche wie das zur Tiefensuche benutzt werden, lediglich der Aufruf DFS-Visit(G, v) muss durch BFS-Visit-Iterative(G, v) ersetzt werden. Die Laufzeit der Breitensuche ist in $\Theta(|V| + |E|)$, denn jeder Knoten wird genau einmal als aktiv markiert und in die Schlange aufgenommen, und wird er der Schlange entnommen, werden genau $\Theta(\deg^+(v) + 1)$ viele Operationen ausgeführt. Da jeder Knoten niemals mehr als einmal in die Schlange aufgenommen wird, ist der Platzbedarf durch $\mathcal{O}(|V|)$ nach oben beschränkt.

Rahmenprogramm

Laufzeit

Extraplatz

5.3 Zusammenhangskomponenten

Gegeben sei ein *ungerichteter* Graph $G = (V, E)$. Das Ziel dieses Abschnitts besteht in der Berechnung der *Zusammenhangskomponenten* von G. Diese sind genau die Äquivalenzklassen der reflexiven und transitiven Hülle von G (siehe Abbildung 5.9). Eine Zusammenhangskomponente ist damit ein Teilgraph $G' = (V', E')$ von G, wobei $E' = \{\{v, w\} \in E \mid v, w \in V'\}$ ist, und in E (also im gegebenen Graphen) keine Kante existiert, die genau einen Knoten in V' und einen Knoten in $V \backslash V'$ hat. Eine Zusammenhangskomponente mit der Knotenmenge V' enthält also alle Kanten zwischen zwei Knoten aus V', die auch der gegebene Graph G enthält, und von einer Zusammenhangskomponente führt keine Kante zu irgendeiner anderen Zusammenhangskomponente. Folglich ist eine Zusammenhangskomponente eindeutig durch ihre Knoten $v \in V'$ festgelegt. Sollen alle Zusammenhangskomponenten eines gegebenen ungerichteten Graphen $G = (V, E)$ bestimmt werden, dann genügt es, eine Partitionierung von V (also eine Zerlegung von V in paarweise disjunkte Mengen $V_1, \ldots, V_k$) zu berechnen, wobei V_i die Knoten einer Zusammenhangskomponente enthält (und für jede Kante in E beide Knoten in der gleichen Menge V_i liegen).

Zusammenhangskomponente

Charakterisierung

Um eine solche Partitionierung zu berechnen, kann einfach der Algorithmus zur Tiefensuche modifiziert werden. Ein Knoten kommt in die gleiche Zusammenhangskomponente wie sein Vorgänger; jedes Mal, wenn die Tiefensuche neu beginnt (im Rahmenprogramm) wird eine neue, leere Zusammenhangskomponente erzeugt. Natürlich kann auch eine Breitensuche zur Berechnung der Zusammenhangskomponenten benutzt werden. Die Laufzeit des so erhaltenen Verfahrens ist in $\Theta(|V|+|E|)$.

Berechnung

Wir beobachten, dass die Anzahl der Äquivalenzklassen exakt der Anzahl der Neustarts im DFS- oder BFS-Rahmenprogramm entspricht. Ebenso hätten wir statt einer Tiefen- oder Breitensuche auch Warshalls Algorithmus benutzen können, dieser hat aber eine erheblich schlechtere Laufzeit von $\Theta(|V|^3)$.

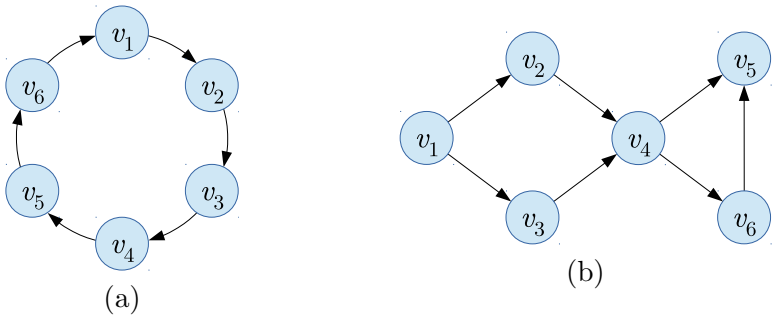


Abb. 5.10 Der Graph K_6 auf der linken Seite (a) kann nicht topologisch sortiert werden. Dagegen ist der Graph auf der rechten Seite (b) kreisfrei und besitzt daher eine topologische Sortierung.

5.4 Topologische Sortierung

MOTIVATION

Gegeben seien eine Menge von Aufgaben $A_1, \ldots, A_n$, von denen einige bereits erledigt sein müssen, bevor andere Aufgaben begonnen werden können. Sind etwa $A_i =$ "Socken anziehen" und $A_j =$ "Schuhe anziehen", dann muss offenbar A_i vor A_j erledigt werden. Gesucht ist eine Reihenfolge, in der die Aufgaben abgearbeitet werden können, sodass jedes Mal, wenn eine Aufgabe A_j begonnen wird, alle für diese Aufgabe vorausgesetzten Aufgaben A_i bereits erledigt wurden.

MODELLIERUNG

Das Problem kann formal durch einen gerichteten Graphen $G = (V, E)$ modelliert werden, der für jede Aufgabe A_i genau einen Knoten enthält, und für jede Aufgabe A_i, die vor einer Aufgabe A_j erledigt sein muss, eine Kante (A_i, A_j) enthält.

TOPOLOGISCHE SORTIERUNG

Gesucht wird eine bijektive Abbildung $\text{ord} : V \to \{1, \ldots, |V|\}$ mit $\text{ord}(v) < \text{ord}(w)$ für alle $(v, w) \in E$. Eine solche Abbildung heisst *topologische Sortierung* von G. Setzen wir $v_i = \text{ord}^{-1}(i)$ für $i = 1, \ldots, |V|$, dann können wir eine topologische Sortierung auch kompakter durch die Sequenz $\langle v_1, \ldots, v_{|V|} \rangle$ repräsentieren. Man beachte, dass die Berechnung einer topologischen Sortierung trotz des gleichen Namens ein anderes Problem als die Sortierung einer Menge von Zahlen ist.

BEISPIEL

Beispiel. *Nicht für jeden Graphen kann eine topologische Sortierung angegeben werden, wie etwa der Graph mit den Knoten v_1 und v_2 sowie den Kanten (v_1, v_2) und (v_2, v_1) zeigt. Das Beispiel kann auf Kreise beliebiger Grösse verallgemeinert werden. Seien $V_n = \{v_1, \ldots, v_n\}$ und $E_n = \{(v_i, v_{i+1}) \mid i \in \{1, \ldots, n-1\}\} \cup \{(v_n, v_1)\}$. Der Graph $K_n = (V_n, E_n)$ (siehe Abbildung 5.10(a)) besitzt keine topologische Sortierung. Sei o.B.d.A. v_1 der Knoten mit $\text{ord}(v_1) = 1$. Damit ord eine gültige topologische Sortierung ist, muss der Wert von ord entlang des Pfades $\langle v_1, \ldots, v_n \rangle$ wachsen. Folglich ist $\text{ord}(v_n) > 1$. Nun ist aber $(v_n, v_1) \in E_n$ mit $\text{ord}(v_n) \not< \text{ord}(v_1)$, also besitzt K_n keine topologische Sortierung.*

Der in Abbildung 5.10(b) dargestellte Graph dagegen ist topologisch sortierbar. Er besitzt sogar zwei topologische Sortierungen, denn sowohl $\langle v_1, v_2, v_3, v_4, v_6, v_5 \rangle$ als auch $\langle v_1, v_3, v_2, v_4, v_6, v_5 \rangle$ sind topologische Sortierungen dieses Graphen.

Das vorige Beispiel zeigte bereits, dass eine topologische Sortierung weder für jeden Graphen existiert, noch eindeutig bestimmt ist. Zur Charakterisierung topologisch sortierbarer Graphen zeigen wir nun das folgende Theorem.

Theorem 5.5. *Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist.*

EXISTENZ EINER TOPOLOGISCHER SORTIERUNG

Beweis. Enthält G einen Kreis, dann besitzt er offenbar keine topologische Sortierung. Wir zeigen nun durch Induktion über die Knotenanzahl n, dass ein Graph topologisch sortiert kann, wenn er kreisfrei ist.

Induktionsanfang ($n = 1$): Ein Graph mit genau einem Knoten v ist topologisch sortierbar, wenn er kreisfrei ist, d.h., keine Schleife besitzt. Setzt man $\text{ord}(v) \leftarrow 1$, dann erhält man eine topologische Sortierung für einen kreisfreien Graphen mit einem Knoten.

Induktionshypothese: Angenommen, jeder kreisfreier Graph mit n Knoten besitzt eine topologische Sortierung.

Induktionsschritt ($n \to n + 1$): Sei G ein Graph mit $n + 1$ Knoten. Wir überlegen zunächst, dass G mindestens einen Knoten v_0 mit Eingangsgrad 0 besitzt. Wäre dies nämlich nicht der Fall, dann könnten wir bei einem Knoten starten und rückwärts laufen (indem wir einer eingehenden Kante zum Vorgänger folgen), und nach spätestens $n + 1$ Schritten hätten wir einen Knoten gefunden, den wir schon früher besucht haben. Damit wäre der Graph also nicht kreisfrei.

Wir entfernen nun v_0 sowie alle von v_0 ausgehenden Kanten aus G. Der so entstandene Graph $G' = (V', E')$ hat $|V'| = n$ Knoten, ist noch immer kreisfrei (denn durch die Entfernung eines Knotens können keine Kreise entstehen) und besitzt daher gemäss Induktionshypothese eine topologische Sortierung $\text{ord}' : V' \to \{1, \ldots, n\}$. Setzen wir nun $\text{ord}(v_0) \leftarrow 1$ sowie $\text{ord}(v) \leftarrow \text{ord}'(v) + 1$ für alle $v \in V'$, dann ist ord eine topologische Sortierung für G, denn für alle Kanten (v_0, w) ist $\text{ord}(v_0) = 1 < \text{ord}(w)$, und für alle Kanten (v, w) mit $v \neq v_0$ ist nach Induktionshypothese $\text{ord}'(v) < \text{ord}'(w)$, folglich also auch $\text{ord}(v) < \text{ord}(w)$. ∎

IDEE FÜR ALGORITHMUS

Der Beweis des vorigen Theorems liefert eine Idee für einen Algorithmus zur Berechnung einer topologischen Sortierung eines gerichteten kreisfreien Graphen G: Wir suchen einen Knoten v_0 mit Eingangsgrad 0 und geben v_0 aus. Danach entfernen wir v_0 sowie alle von v_0 ausgehenden Kanten aus G und wenden das Verfahren rekursiv auf dem verbleibenden Graphen an, bis dieser entweder überhaupt keine oder keine Knoten mit Eingangsgrad 0 mehr besitzt. Im ersten Fall hätten wir eine topologische Sortierung gefunden, im zweiten Fall hätten wir einen Kreis gefunden und es gäbe keine topologische Sortierung. Das Problem dieses Algorithmus besteht in der Laufzeit: Zum Finden eines Knotens mit Eingangsgrad 0 können wir bei einem beliebigen Knoten von G starten und jeweils den eingehenden Kanten rückwärts folgen, bis ein geeigneter Kandidat gefunden wird. Im schlimmsten Fall müssen wir aber den ganzen Graphen rückwärts entlanglaufen und benötigen $\Omega(|V|)$ Schritte, was zu einem Algorithmus mit Laufzeit $\Omega(|V|^2)$ führt.

VERBESSERTE IDEE

Die Idee zur Verbesserung besteht nun darin, die Eingangsgrade aller Knoten im Voraus zu berechnen und in einem Array A zu speichern, und alle Knoten mit Eingangsgrad 0 auf einen Stapel S zu legen. Danach entfernen wir den obersten Knoten v von S, geben v aus (bzw. ordnen v seine korrekte Position in der topologischen Sortierung zu) und dekrementieren den Eingangsgrad aller Knoten w mit

$(v, w) \in E$. Falls dabei ein Knoten mit einem neuen Eingangsgrad von 0 entdeckt wird, so wird dieser auf S gelegt. Dieses Verfahren wird wiederholt, bis S leer ist.

ALGORITHMUS TOPLOGICAL-SORT(V, E)

```
 1 S ← EmptyStack                          ▷ Stapel S initialisieren
 2 for each v ∈ V do A[v] ← 0
 3 for each (v, w) ∈ E do A[w] ← A[w] + 1  ▷ Berechne Eingangsgrade
 4 for each v ∈ V do                       ▷ Lege Knoten mit Eingangs-
 5     if A[v] = 0 then Push(v, S)         ▷ grad 0 auf den Stapel S
 6 i ← 1
 7 while S not empty do
 8     v ← Pop(S)                          ▷ Knoten mit Eingangsgrad 0
 9     ord[v] ← i; i ← i + 1               ▷ Weise korrekte Position zu
10     for each (v, w) ∈ E do              ▷ Verringere Eingangsgrad
11         A[w] ← A[w] − 1                 ▷ der Nachfolger
12         if A[w] = 0 then Push(w, S)
13 if i = |V| + 1 then return ord else "Graph enthält einen Kreis"
```

LAUFZEIT UND KORREKTHEIT, TEIL I

Theorem 5.6. *Sei $G = (V, E)$ ein gerichteter kreisfreier Graph. Der Algorithmus* TOPOLOGICAL-SORT(V, E) *berechnet in Zeit $\Theta(|V| + |E|)$ eine topologische Sortierung ord für G.*

Beweis. Die Korrektheit des Vorgehens folgt im Wesentlichen aus Theorem 5.5 zusammen mit der Beobachtung, dass für kreisfreie Graphen das Dekrementieren der Knotengrade aller Nachbarknoten von v in den Schritten 10–11 exakt der Entfernung von v entspricht. Es verbleibt zu zeigen, dass am Ende eine topologische Sortierung ord zurückgegeben wird. Ein Knoten v wird nur dann in S eingefügt, wenn er entweder in G einen Eingangsgrad von 0 hat, oder wenn allen Knoten $u \in V$ mit $(u, v) \in E$ bereits früher ein Wert $\text{ord}[u] > 0$ zugewiesen wurde. Wird also ein Knoten v aus S entfernt, dann wird $\text{ord}[v]$ auf einen Wert gesetzt, der grösser als alle $\text{ord}[u]$, $(u, v) \in E$, ist. Ausserdem wird v kein weiteres Mal in S eingefügt. Folglich ist ord eine topologische Sortierung, und nach Terminierung der Schleife in Schritt 6 ist $i = |V|$. Somit wird ord zurückgegeben.

Die Schleifen in den Schritten 2 und 4 haben eine Laufzeit von exakt $\Theta(|V|)$, die Schleife in Schritt 3 hat eine Laufzeit von $\Theta(|E|)$. Wir haben bereits gesehen, dass die Schleife in Schritt 7 genau $|V|$ Mal durchlaufen wird. Ausserdem wird in den Schritten 7–12 jede Kante $(v, w) \in E$ nur genau einmal betrachtet. Also beträgt die Laufzeit der Schritte 7–12 genau $\Theta(|V| + |E|)$, was somit der gesamten Laufzeit des Algorithmus entspricht. ∎

LAUFZEIT UND KORREKTHEIT, TEIL II

Theorem 5.7. *Sei $G = (V, E)$ ein gerichteter,* nicht *kreisfreier Graph. Der Algorithmus* TOPOLOGICAL-SORT(V, E) *terminiert nach $\Theta(|V| + |E|)$ Schritten und gibt "Graph enthält einen Kreis" zurück.*

Beweis. Sei $K = \langle v_1, \ldots, v_k, v_1 \rangle$ ein in G enthaltener Kreis. In jedem Durchlauf der Schleife in Schritt 7 bleibt $A[v_i] \geq 1$ für alle $i \in \{1, \ldots, k\}$, folglich wird *keiner* der Knoten $v_1, \ldots, v_k$ jemals in S eingefügt. Somit wird die Schleife in Schritt 7 maximal $|V| - k < |V|$ Mal durchlaufen, am Ende ist $i < |V| + 1$ und es wird in Schritt 13 keine topologische Sortierung zurückgegeben. Der Beweis der Laufzeit verläuft analog zu Theorem 5.6 mit der Änderung, dass die Schleife in Schritt 7 nicht exakt $|V|$ Mal, sondern höchstens $|V| - 2$ Mal durchlaufen wird. Man beachte, dass die Gesamtlaufzeit in $\Theta(|V| + |E|)$ bleibt, da die Schritte 2–5 immer so viel Zeit benötigen. ∎

5.5 Kürzeste Wege

MOTIVATION

Bereits zu Beginn des Kapitels wurde die Berechnung einer schnellsten Route in einem gegebenen Strassennetz als Beispiel für ein Problem genannt, das mithilfe der Graphentheorie gelöst werden kann. Zur Berechnung einer schnellsten Zugverbindung zwischen zwei gegebenen Städten kommen zusätzlich noch Abfahrtszeiten hinzu, was das Problem etwas komplexer macht, aber dennoch kann auch dieses Problem als graphentheoretisches Problem formuliert werden. Beim *Kürzeste-Wege-Problem* ist als Eingabe ein gerichteter, gewichter Graph $G = (V, E, w)$ gegeben, wobei wie zuvor $w : E \to \mathbb{R}$ die Kantengewichtungsfunktion ist. Weiterhin sind zwei Knoten $s, t \in V$ gegeben. Gesucht ist ein Weg $W = \langle s = v_0, \ldots, v_k = t \rangle$ von s nach t, dessen Kosten $c(W) = \sum_{i=1}^{k} w((v_{i-1}, v_i))$ minimal unter allen Wegen von s nach t ist. Wir nennen W dann vereinfachend auch einen *kürzesten Weg*, wobei sich diese Bezeichnung auf die Kantengewichte und im Allgemeinen nicht auf die Anzahl der Kanten auf dem Weg bezieht. Vereinfachend nehmen wir an, dass es mindestens einen Weg gibt, der in s startet und in t endet. Ob dies der Fall ist, kann im Vorfeld leicht mit einer in s startenden Breitensuche festgestellt werden. Falls kein einziger st-Weg existiert, geben wir eine Fehlermeldung aus und sind fertig.

KÜRZESTE-WEGE-PROBLEM

VEREINFACHUNG

Wir müssen zunächst überlegen, ob das soeben formulierte Kürzeste-Wege-Problem überhaupt sinnvoll ist. Es kann nämlich unendlich viele Wege von s nach t geben, und es kann sogar passieren dass man zu jedem solchen st-Weg einen weiteren, kürzeren st-Weg findet (vgl. Abbildung 5.11). Solch eine Situation kann auftreten, wenn es Kreise mit negativem Gewicht gibt. Unproblematisch dagegen ist es, wenn alle Kantengewichte nicht-negativ (also grösser oder gleich Null) sind, denn dann hat jeder Kreis ein nicht-negatives Gewicht. Es gibt also auf jeden Fall einen kürzesten Weg, der keinen Knoten mehrfach besucht, und der folglich ein Pfad ist. Ein Pfad hat höchstens $|V|$ viele Knoten, also gibt es für einen festen Graphen nur endlich

EXISTENZ VON KÜRZESTEN WEGEN

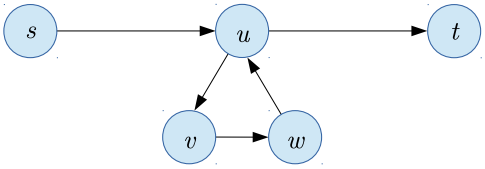


Abb. 5.11 Am Knoten u gibt es einen Kreis, also gibt es unendlich viele Wege von s nach t. Wenn die Summe der Kantengewichte auf dem Kreis $\langle u, v, w, u \rangle$ negativ ist, gibt es keinen kürzesten st-Weg, denn zu jedem solchen Weg findet man einen kürzeren st-Weg, indem der Kreis einmal mehr durchlaufen wird.

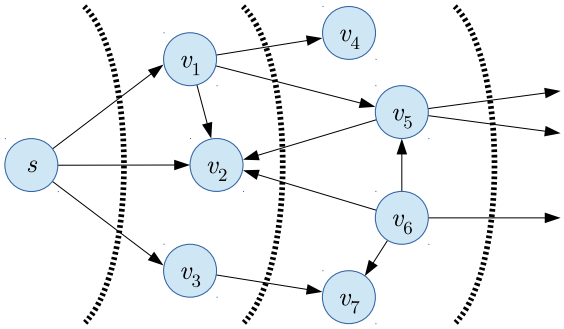


Abb. 5.12 Eine Breitensuche auf einem Beispielgraphen. Die dick gestrichelten Linien unterteilen den Graphen in Schichten, und alle Knoten einer Schicht sind gleich weit von s entfernt: Ein kürzester Weg von s zu s hat Länge 0, ein kürzester Weg von s zu v_1, v_2 und v_3 hat Länge 1, und ein kürzester Weg von s zu v_4, v_5, v_6 und v_7 hat Länge 2.

viele Pfade. Unter diesen muss einer kürzestmöglich (in Hinblick auf sein Gewicht) sein. Wir werden jetzt zuerst überlegen, wie ein kürzester Weg in einem Graphen mit nicht-negativen Kantengewichten berechnet werden kann. Danach diskutieren wir, wie man auch negative Kantengewichte zulassen kann, und wie Kreise mit negativem Gewicht behandelt werden können.

5.5.1 Kürzeste Wege in Graphen mit uniformen Kantengewichten

Wenn alle Kanten die gleichen Kosten (z.B. 1) haben, dann können wir einen kürzesten Weg einfach mit einer modifizierten Breitensuche von s aus finden. Dazu speichern wir für jeden Knoten $v \in V$, von welchem Knoten $p[v]$ aus der Knoten v entdeckt wurde. Der Wert $p[v]$ gibt also den Vorgänger (*engl.* predecessor) von v auf einem kürzesten Weg von s nach v an, und wird für alle $v \in V$ mit **null** initialisiert (zu Beginn wurde noch kein Knoten entdeckt). Damit ergibt sich der folgende Algorithmus:

Iterative Breitensuche

BFS-Visit-Iterative(G, v)

```
1 for each v ∈ V do p[v] = null          ▷ Initialisiere Vorgänger
2 Q ← ∅                                  ▷ Initialisiere leere Schlange Q
3 Enqueue(s, Q)                          ▷ Füge s zur Schlange Q hinzu
4 while Q ≠ ∅ do
5     u ← Dequeue(Q)                     ▷ Aktueller Knoten
6     for each (u, v) ∈ E do
7         if p[v] ≠ null then            ▷ Falls v noch nicht entdeckt wurde
8             p[v] ← u                   ▷ Speichere u als Vorgänger von v
9             Enqueue(v, Q)              ▷ Füge v zur Schlange Q hinzu
```

Laufzeit

Die Laufzeit ist natürlich noch immer in $\Theta(|V| + |E|)$, da pro Knoten und pro Kante nur ein konstanter Extraaufwand betrieben wird. Um nun den kürzesten Weg von s zu t zu rekonstruieren, setzen wir zunächst $v \leftarrow t$ und geben v aus. Danach setzen wir $v \leftarrow p[v]$ und geben v aus, bis $v = s$ gilt. Auf diese Art wird ein kürzester Weg von s nach t rückwärts rekonstruiert (wir finden also zuerst den letzten Knoten auf diesem Weg, dann den vorletzten, usw). Man beachte, dass dieser Weg

Rekonstruktion des Weges

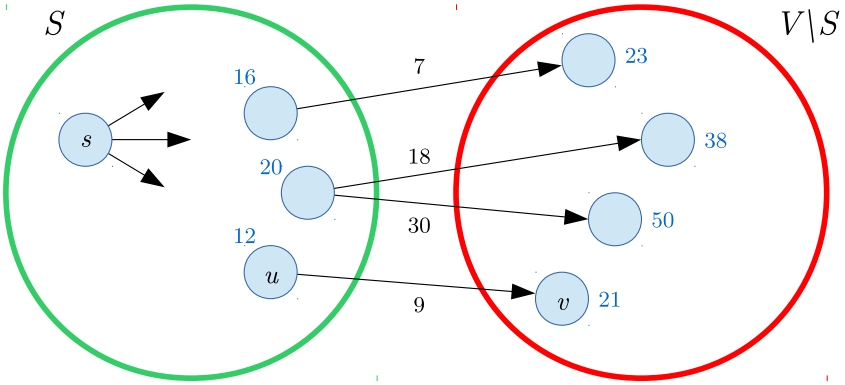


Abb. 5.13 Unter allen Kanten (u, v), die von S zu $V\backslash S$ verlaufen, wird diejenige gewählt, für die $d[u]+w((u,v))$ minimal ist. Die blauen Zahlen neben einem Knoten geben $d[u]$ (links) bzw. $d[u]+w((u,v))$ (rechts) an, die schwarzen Zahlen oberhalb bzw. unterhalb der Kanten die entsprechenden Gewichte $w((u,v))$. Gewählt wird die Kante mit Gewicht 9.

kreisfrei ist, also aus höchstens $|V|$ Knoten besteht. Damit ist die für die Rekonstruktion benötigte Zeit in $\mathcal{O}(|V|)$ und im Vergleich zur Laufzeit der Breitensuche selbst vernachlässigbar.

Das ursprüngliche Ziel war die Berechnung eines kürzesten Weges von s nach t. Tatsächlich macht die Breitensuche aber etwas mehr, denn sie berechnet die kürzesten Wege vom Startknoten s zu allen Knoten des Graphen! Ausserdem können wir mit nur einem Durchlauf feststellen, welche Knoten nicht von s aus erreicht werden können. Das sind genau die Knoten v mit $p[v] = \textbf{null}$.

5.5.2 Kürzeste Wege in Graphen mit nicht-negativen Kantengewichten

Die im vorigen Unterabschnitt vorgestellte Breitensuche kann nun verallgemeinert werden, um alle kürzesten Wege (bzw. ihre Längen) vom Startknoten s zu allen Knoten $v \in V$ in einem Graphen mit allgemeinen, nicht-negativen Kantengewichten zu berechnen. Wir erlauben zunächst die vereinfachende Annahme, dass von s aus jeder Knoten $v \in V$ erreicht werden kann. Diese Annahme vereinfacht später die Analyse und stellt keine Einschränkung dar, denn wie gesehen können wir eine Breitensuche von s aus starten um alle von dort aus erreichbaren Knoten zu finden; alle übrigen Knoten werden entsprechend als "unerreichbar" markiert.

VEREINFACHUNG

Algorithmengerüst Unsere verallgemeinerte Breitensuche arbeitet induktiv und verwaltet für jeden Knoten $v \in V$ einen Wert $d[v]$, der angibt, wie lang ein kürzester Weg von s zu v höchstens ist. Initial setzen wir $d[s] \leftarrow 0$, und für alle anderen $v \in V\backslash\{s\}$ setzen wir $d[v] \leftarrow \infty$. Bei der Terminierung des Algorithmus soll $d[v]$ genau die Länge eines kürzesten Weges von s nach v angeben (und nicht nur eine Abschätzung). Der Algorithmus verwaltet ausserdem eine Menge von Knoten S, für die bereits bekannt ist, dass $d[v]$ genau der Länge eines kürzesten Weges von s nach v entspricht. Initial enthält S nur s mit $d[s] = 0$. Nun wird S in jedem Schritt um genau einen Knoten erweitert. Dazu wählen wir unter allen Kanten $(u, v) \in (S \times (V\backslash S)) \cap E$ (also allen Kanten (u, v), die von einem Knoten in S zu einem Knoten in $V\backslash S$ verlaufen) diejenige, für die $d[u] + w((u, v))$ minimal ist (siehe Abbildung 5.13). Dann setzen wir $d[v] \leftarrow d[u]+w((u, v))$ und fügen S den Knoten v hinzu. Der Algorithmus endet, sobald S alle Knoten enthält.

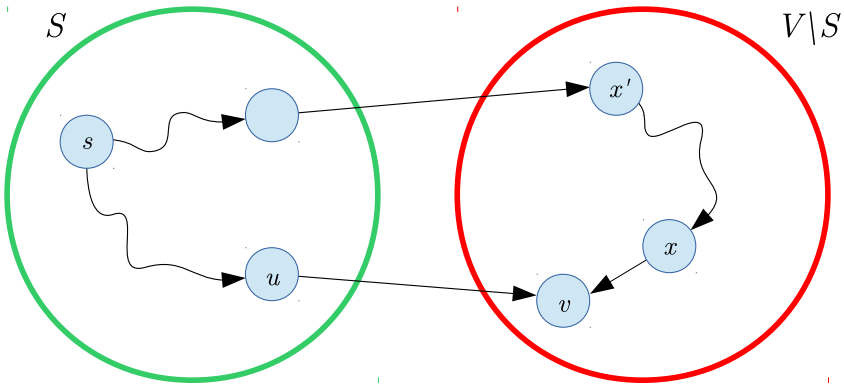


Abb. 5.14 Der kürzeste Weg von s nach x ist mindestens so lang wie ein kürzester Weg von s nach v über u. Ein sv-Weg, der zusätzlich noch x' und x besucht, ist daher nicht kürzer als der kürzeste Weg von s nach v über u.

Korrektheit Zum Nachweis der Korrektheit führen wir eine vollständige Induktion über die Anzahl der Knoten in S aus, um zu zeigen, dass $d[v]$ für alle $v \in S$ stets genau die Länge eines kürzesten Weges von s nach v angibt.

Induktionsanfang ($|S| = 1$): Zu Beginn enthält S nur den Startknoten s mit $d[s] = 0$, folglich ist die Aussage wahr.

Induktionshypothese: Angenommen, für jeden Knoten $v \in S$ entspricht $d[v]$ der Länge eines kürzesten Weges von s nach v.

Induktionsschritt ($|S| \to |S| + 1$): Der Algorithmus wählt eine Kante $(u, v) \in (S \times (V \backslash S)) \cap E$, für die $d[u] + w((u, v))$ minimal ist, setzt $d[v] \leftarrow d[u] + w((u, v))$ und fügt v der Menge S hinzu. Wir argumentieren jetzt, dass $d[v]$ tatsächlich die Länge eines kürzesten Weges von s nach v angibt. Dazu betrachten wir einen kürzesten Weg von s nach v. Sei nun x der Knoten, den dieser Weg direkt vor v besucht. Liegt x in S, dann gilt

$$d[v] = d[u] + w((u, v)) \leq$$
$$d[x] + w((x, v)) = \text{Länge eines kürzesten Weges von } s \text{ nach } v. \quad (108)$$

Man beachte, dass sowohl x als auch u in S liegen, folglich geben $d[u]$ bzw. $d[x]$ die Längen der kürzesten Wege von s nach u bzw. x an. Die Ungleichung gilt, weil sowohl (u, v) als auch (x, v) von S nach $V \backslash S$ verlaufen und unter all diesen Kanten (u, v) diejenige ist, für die $d[u] + w((u, v))$ minimal ist. Da es ausserdem einen Pfad von s nach u sowie eine Kante von u nach v gibt, gilt sogar Gleichheit.

Liegt dagegen x nicht in S, dann enthält jeder Weg von s nach v mindestens eine Kante, die von einem Knoten in S zu einem Knoten $x' \in V \backslash S$ verläuft. Also hat (wie zuvor) jeder Weg von s nach x' mindestens Länge $d[u]+w((u, v))$, und jeder Weg von s nach x natürlich erst recht (da es keine Kanten mit strikt negativem Gewicht gibt). Daher muss auch der Weg von s nach v über x mindestens Länge $d[u] + w((u, v))$ haben (siehe Abbildung 5.14). Andererseits hat der Weg von s nach v über u genau die Länge $d[u] + w((u, v))$. Daher kann kein Weg über einen Knoten $x \in V \backslash S$ kürzer sein als der Weg über u, also gibt $d[v] = d[u] + w((u, v))$ wie gewünscht die Länge eines kürzesten Weges von s nach v an. ∎

Präzisierung des Algorithmus Es verbleibt zu überlegen, wie die Auswahl einer geeigneten Kante von S nach $V\backslash S$ realisiert werden kann. Wir könnten zum Beispiel einfach über alle Kanten $(u, v) \in E$ des Graphen iterieren und prüfen, ob $u \in S$ und $v \notin S$ gilt. Unter allen Kanten mit dieser Eigenschaft merken wir uns diejenige, für die $d[u] + w((u, v))$ minimal ist. In dieser einfachen Realisierung benötigt jeder Schritt Zeit $\mathcal{O}(|E|)$, da über alle Kanten iteriert wird. Da in jedem Schritt $|S|$ um 1 wächst (und niemals grösser als $|V|$ wird), erhalten wir insgesamt einen Algorithmus mit Laufzeit $\mathcal{O}(|V||E|)$.

Triviale Variante

Der niederländische Informatiker Dijkstra beobachtete, dass es nicht notwendig ist, in jedem Schritt alle Kanten zu betrachten. Stattdessen genügt es, sich für jeden Knoten v eine vorläufige obere Schranke $d[v]$ zu merken und entsprechend zu aktualisieren, sobald eine neue Kante zu v gefunden wird. Sobald also ein Knoten u der Menge S hinzugefügt wird, iterieren wir über alle von u ausgehenden Kanten $(u, v) \in E$, prüfen ob $d[u] + w((u, v)) < d[v]$ gilt, und setzen in dem Fall $d[v]$ auf $d[u] + w((u, v))$ (denn wir haben einen kürzeren Weg zu v entdeckt). Gilt dagegen $d[u] + w((u, v)) \geq d[v]$, dann ist der Weg von s zu v über u nicht kürzer als der bisher beste gefundene Weg von s zu v. In jedem Schritt wählen wir dann als nächstes den Knoten $v \in V\backslash S$, für den $d[v]$ minimal ist, fügen ihn S hinzu und verfahren wie soeben beschrieben. In dieser Realisierung wird jede Kante $(u, v) \in E$ insgesamt genau einmal angeschaut (genau dann, wenn u der Menge S hinzugefügt wird), und die Auswahl eines geeigneten Knotens v kann in Zeit $\mathcal{O}(|V|)$ geschehen, da die Menge $V\backslash S$ höchstens so viele Knoten hat. Da dieser Auswahlschritt $|V|$ Mal ausgeführt wird, erhalten wir insgesamt einen Algorithmus mit Laufzeit $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$, was gegenüber der trivialen Variante schon eine Verbesserung darstellt.

Dijkstras Algorithmus (original)

Laufzeit

Die Laufzeit der zuvor beschriebenen ursprünglichen Variante von Dijkstras Algorithmus ist quadratisch in der Anzahl der Knoten, da in jedem Schritt über alle Knoten in $V\backslash S$ iteriert wird, um den Knoten v zu finden, für den $d[v]$ minimal ist. Eine Verbesserung besteht nun darin, die Knoten in $V\backslash S$ mit einer Prioritätswarteschlange zu verwalten (siehe Abschnitt 4.1) und als Priorität eines Knotens den aktuellen Wert $d[v]$ zu verwenden. Mit einer geeigneten Datenstruktur kann dann der Knoten v mit minimalem Wert $d[v]$ schneller gefunden werden. Anders als früher wollen wir jetzt aber einen Knoten mit *minimaler* Priorität finden. Ausserdem erlauben wir, dass die Prioritätswarteschlange mehrere Knoten mit gleicher Priorität enthält, denn es kann ja zwei verschiedene Knoten $v, w \in V\backslash S$ mit $d[v] = d[w]$ geben. Da der Wert $d[v]$ auch nachträglich noch absinken kann, benötigen wir weiterhin eine Operation, die dies umsetzt. Konkret muss unsere Prioritätswarteschlange Q also die folgenden drei Operationen unterstützen:

Dijkstras Algorithmus mit Heaps

Operationen

- Extract-Min(Q): Liefert *irgendeinen* Knoten v zurück, dessen Priorität unter allen in Q verwalteten Knoten minimal ist, und entfernt ihn aus Q. Die Operation darf nur dann aufgerufen werden, wenn Q nicht leer ist.
- Insert(v, π, Q): Fügt den Knoten v mit Priorität $\pi \in \mathbb{Q}_0^+$ in Q ein. Die Operation darf nur aufgerufen werden, wenn Q den Knoten v noch nicht enthält.
- Decrease-Key(v, π, Q): Setzt die Priorität des Knotens v auf π herab. Die Operation darf nur aufgerufen werden, wenn Q den Knoten v enthält und die Priorität von v vor Aufruf der Operation grösser oder gleich π ist.

Der Algorithmus kann nun wie folgt formuliert werden:

Dijkstras Algorithmus

DIJKSTRA$(G = (V, E), s)$

```
 1 for each v ∈ V\{s} do                          ▷ Initialisiere für alle Knoten die
 2     d[v] ← ∞; p[v] ← null                      ▷ Distanz zu s sowie Vorgänger
 3 d[s] ← 0; p[s] ← null                          ▷ Initialisierung des Startknotens
 4 Q ← ∅                                          ▷ Leere Prioritätswarteschlange Q
 5 INSERT(s, 0, Q)                                ▷ Füge s zu Q hinzu
 6 while Q ≠ ∅ do
 7     u ← EXTRACT-MIN(Q)                         ▷ Aktueller Knoten
 8     for each (u, v) ∈ E do                     ▷ Inspiziere Nachfolger
 9         if p[v] = null then                    ▷ v wurde noch nicht entdeckt
10             d[v] ← d[u] + w((u, v))            ▷ Berechne obere Schranke
11             p[v] ← u                           ▷ Speichere u als Vorgänger von v
12             ENQUEUE(v, d[v], Q)                ▷ Füge v zu Q hinzu
13         else if d[u] + w((u, v)) < d[v] then   ▷ Kürzerer Weg zu v entdeckt
14             d[v] ← d[u] + w((u, v))            ▷ Aktualisiere obere Schranke
15             p[v] ← u                           ▷ Speichere u als Vorgänger von v
16             DECREASE-KEY(v, d[v], Q)           ▷ Setze Priorität von v herab
```

Rekonstruktion des Weges

Wie bei der Breitensuche im vorigen Unterabschnitt 5.5.1 bezeichnet $p[v]$ den Vorgängerknoten von v auf einem kürzesten Weg von s nach v. Dieser Vorgängerknoten kann auch benutzt werden, um festzustellen, ob ein Knoten $v \in V\backslash S$ bereits in Q gespeichert ist: Wenn $p[v] =$ **null** ist, dann wurde v noch nie in Q eingefügt. Der Algorithmus berechnet für einen gegebenen Startknoten s die Längen der kürzesten Wege zu allen Knoten des Graphen sowie die entsprechenden Vorgängerknoten. Der kürzeste Weg selbst von s zu einem beliebigen Knoten t kann wieder genau wie im vorigen Unterabschnitt beschrieben rekonstruiert werden.

Laufzeit

Theorem 5.8. *Wird Dijkstras Algorithmus mit Heaps implementiert, dann ist die Laufzeit in* $\mathcal{O}((|V| + |E|) \log |V|)$.

Beweis. Der in Abschnitt 2.4 vorgestellte Heap kann so modifiziert werden, dass alle drei Operationen EXTRACT-MIN, INSERT und DECREASE-KEY unterstützt werden und nur Zeit $\mathcal{O}(\log n)$ benötigen, wobei n die Anzahl der von Q verwalteten Schlüssel angibt. Da Q jeden Knoten aus V höchstens einmal enthält, ist die Laufzeit aller drei Operationen also stets in $\mathcal{O}(\log |V|)$.

Die Schritte 1–5 des Algorithmus laufen in Zeit $\mathcal{O}(|V|)$. Die Schritte 9–16 brauchen insgesamt Zeit höchstens $\mathcal{O}(\log |V|)$. Zentral für die Laufzeitanalyse ist die Beobachtung, dass jeder Knoten genau einmal in Q eingefügt und genau einmal aus Q entfernt wird. Wurde ein Knoten u aus Q entfernt, dann wird er niemals wieder in Q eingefügt, und sein Wert $d[u]$ wird niemals wieder verändert. Der Grund ist, dass für alle später aus Q entfernten Knoten x der Wert $d[x]$ mindestens so gross wie $d[u]$ ist; folglich ist der Vergleich in Schritt 13 niemals erfüllt. Da jeder Knoten nur genau einmal aus Q entfernt wird, wird jede von u ausgehende Kante $(u, v) \in E$ im Verlauf des Algorithmus nur genau einmal angeschaut. Die Gesamtlaufzeit beträgt damit $\mathcal{O}((|V| + |E|) \log |V|)$. ∎

Der soeben vorgestellte Algorithmus kann noch ein wenig verbessert werden. Dazu wird die Prioritätswarteschlange Q durch einen sogenannten *Fibonacci-Heap* (ohne weitere Erklärung) implementiert. Dieser erlaubt die Hinzufügung eines neuen Knotens in Zeit $\mathcal{O}(1)$, die Absenkung der Priorität eines Knotens in amortisierter Zeit $\mathcal{O}(1)$ sowie die Extraktion eines Knotens mit minimaler Priorität in amortisierter Zeit $\mathcal{O}(\log n)$ (wobei wie zuvor n die Anzahl der in Q gespeicherten Knoten angibt). Da jeder Knoten genau einmal in Q eingefügt und genau einmal aus Q entfernt wird, fällt für diese beiden Operationen insgesamt Zeit $\mathcal{O}(|V|\log|V|)$ an. Zusätzlich müssen insgesamt höchstens $|E|$ viele Prioritäten abgesenkt werden, was in Zeit $\mathcal{O}(|E|)$ realisiert werden kann. Wird also Dijkstras Algorithmus mit einem Fibonacci-Heap implementiert, dann beträgt die Laufzeit nur noch $\mathcal{O}(|V|\log|V|+|E|)$. Dies ist auch die Algorithmenvariante, die üblicherweise als *Dijkstras Algorithmus* bezeichnet wird (auch wenn der ursprünglich von Dijkstra vorgeschlagene Algorithmus keine Fibonacci-Heaps verwendete und daher eine schlechtere Laufzeit aufwies). In praktischen Anwendungen spielen Fibonacci-Heaps allerdings keine grosse Rolle, da die in der $\mathcal{O}$-Notation seiner Laufzeit versteckten Konstanten recht gross sind. Herkömmliche Heaps stellen daher oftmals eine bessere Wahl dar.

DIJKSTRAS ALGORITHMUS MIT FIBONACCI-HEAPS

PRAKTISCHE RELEVANZ

5.5.3 Kürzeste Wege in Graphen mit allgemeinen Kantengewichten

Wir haben bereits überlegt, dass die Situation in Graphen mit allgemeinen Kantengewichten schwieriger ist, da es Kreise mit negativem Gewicht geben kann (vgl. Abbildung 5.11). Bevor wir einen Algorithmus für dieses allgemeine Kürzeste-Wege-Problem angeben, überlegen wir zunächst, welche der bisherigen Erkenntnisse weiterverwendet werden können. Wie zuvor sei s der Startknoten, von dem ausgehend die kürzesten Wege zu allen anderen Knoten des Graphen berechnet werden sollen.

PROBLEM

Für jede Kante $(u,v) \in E$ gilt nach wie vor $d[v] \leq d[u] + w((u,v))$, denn ein kürzester sv-Weg ist höchstens so lang wie ein kürzester su-Weg und dem Gewicht (bzw. der Länge) der Kante (u,v) zusammen. Wenn wir also auf eine Kante (u,v) mit $d[v] > d[u] + w((u,v))$ treffen, können wir wie zuvor $d[v] \leftarrow d[u] + w((u,v))$ setzen. Wir sagen dann auch, die Kante (u,v) werde *relaxiert*. Wird eine Kante (u,v) mit $d[v] \leq d[u] + w((u,v))$ relaxiert, dann bleibt $d[v]$ natürlich unverändert, da der sv-Weg über u keine Verbesserung gegenüber dem bisher besten Weg darstellt. Fords Algorithmus relaxiert in jedem Schritt jede Kante des Graphen genau einmal und beginnt dann wieder von vorn, bis sich in einem Schritt kein Wert $d[v]$ mehr ändert. Das Problem dieses Verfahrens ist, dass es niemals endet, wenn der Graph Kreise mit negativem Gewicht hat.

RELAXIERUNG

FORDS ALGORITHMUS

Wir haben bereits früher überlegt, dass jeder kürzeste Weg, sofern existent, ein Pfad sein muss. Ein kürzester Weg besucht also keinen Knoten mehr als einmal. Dies gilt natürlich auch dann noch, wenn ein Graph Kanten mit negativen Gewichten hat. Der US-amerikanische Mathematiker Bellman schlug ein dynamisches Programm zur Berechnung aller kürzesten Wege von einem gegebenen Startknoten aus vor. Dazu wird eine $|V| \times |V|$-Tabelle benutzt, und ein Eintrag $T[v,i]$ (mit $v \in V$ und $i \in \{0,\ldots,|V|-1\}$) gibt die Länge eines kürzesten Weges von s nach v mit höchstens i Kanten an. Initial setzen wir $T[s,0] \leftarrow 0$ und $T[v,0] \leftarrow \infty$ für alle $v \in V \backslash \{s\}$. Im i-ten Schritt rechnen wir

ALGORITHMUS VON BELLMAN

$$T[v,i] \leftarrow \min\Big(T[v][i-1], \min_{(u,v)\in E}\big(T[u][i-1] + w((u,v))\big)\Big), \tag{109}$$

denn ein kürzester sv-Weg mit höchstens i Kanten wurde entweder bereits im $(i-1)$-ten Schritt gefunden, oder es gibt einen kürzesten Weg mit höchstens $i-1$ Kanten zu einem Knoten u sowie eine Kante von u nach v. Nachdem die gesamte Tabelle ausgefüllt wurde, setzen wir $d[v] \leftarrow T[v, n-1]$, iterieren über alle Kanten $(u,v) \in E$ und prüfen, ob noch eine Kante relaxiert werden kann. Falls ja, dann gibt es einen Kreis mit negativem Gewicht. Ansonsten können die kürzesten Wege einfach aus der DP-Tabelle abgelesen werden.

Laufzeit

Ein Tabelleneintrag kann in Zeit $\mathcal{O}(|V|)$ berechnet werden (denn er hängt von höchstens $|V|$ anderen Tabelleneinträgen ab), und die Tabelle hat $|V|^2$ viele Einträge. Nach der Berechnung der Tabelle kann in Zeit $\mathcal{O}(|E|)$ festgestellt werden, ob es einen Kreis mit negativem Gewicht gibt. Die Gesamtlaufzeit von Bellmans Algorithmus ist also in $\mathcal{O}(|V|^3 + |E|) = \mathcal{O}(|V|^3)$.

Man kann die Algorithmen von Ford und Bellman kombinieren und sich überlegen, dass jede Kante $(u,v) \in E$ niemals mehr als $|V|-1$ Mal relaxiert werden muss. Führt eine weitere Relaxierung von (u,v) dann dazu, dass sich der Wert $d[v]$ ändert, dann enthält der Graph einen Kreis mit negativem Gewicht. Diese Idee macht sich der Algorithmus von Bellman und Ford zunutze, der wie folgt funktioniert.

Algorithmus von Bellman und Ford

Bellman-Ford$(G = (V,E), s)$

```
 1 for each v ∈ V\{s} do                    ▷ Initialisiere für alle Knoten die
 2     d[v] ← ∞; p[v] ← null                ▷ Distanz zu s sowie Vorgänger
 3 d[s] ← 0; p[s] ← null                    ▷ Initialisierung des Startknotens
 4 for i ← 1, 2, ..., |V| − 1 do            ▷ Wiederhole |V| − 1 Mal
 5     for each (u, v) ∈ E do               ▷ Iteriere über alle Kanten (u, v)
 6         if d[v] > d[u] + w((u, v)) then  ▷ Relaxiere Kante (u, v)
 7             d[v] ← d[u] + w((u, v))      ▷ Berechne obere Schranke
 8             p[v] ← u                     ▷ Speichere u als Vorgänger von v
 9 for each (u, v) ∈ E do                   ▷ Prüfe, ob eine weitere Kante
10     if d[u] + w((u, v)) < d[v] then      ▷ relaxiert werden kann
11         Melde Kreis mit negativem Gewicht
```

Sofern kein Kreis mit negativem Gewicht existiert, kann ein kürzester Pfad von s zu einem beliebigen Zielknoten t wie in den vorigen Unterabschnitten beschrieben rekonstruiert werden, indem man dem Vorgänger $p[v]$ beginnend mit $v = t$ folgt, bis der Startknoten erreicht wurde.

Laufzeit

Die Initialisierung (Schritte 1–3) läuft in Zeit $\mathcal{O}(|V|)$. Die Berechnung der Werte $d[v]$ (Schritte 4–8) läuft in Zeit $\mathcal{O}(|V||E)$, und wie zuvor kann in Zeit $\mathcal{O}(|E|)$ festgestellt werden, ob ein Kreis mit negativem Gewicht existiert (Schritte 9–11). Die Gesamtlaufzeit des Algorithmus von Bellman und Ford ist also in $\mathcal{O}(|V||E|)$, was eine Verbesserung gegenübers Bellmanns originalem dynamischen Programm darstellt.

5.5.4 Kürzeste Wege zwischen allen Paaren von Knoten

In einem gewichteten Graphen $G = (V,E,w)$ können wir für jeden Knoten messen, wie oft er auf einem kürzesten Weg zwischen zwei Knoten vorkommt. Knoten, die auf vielen kürzesten Wegen vorkommen, kommt oftmals eine herausragende Bedeutung zu. Dieses Mass ist unter dem Namen *Betweenness Centrality* bekannt. Zur Berechnung ist es erforderlich, alle kürzesten Pfade zwischen allen Knoten des Graphen

Betweenness Centrality

zu berechnen. Zur Lösung dieses Problems könnte zum Beispiel Dijkstras Algorithmus verwendet werden, der dann von jedem Knoten $s \in V$ aus gestartet wird. Dies benötigt insgesamt Zeit $\mathcal{O}(|V|^2 \log |V| + |V||E|)$ und funktioniert natürlich nur, wenn alle Kantengewichte nicht-negativ sind. Im Falle allgemeiner Kantengewichtsfunktionen kann der Algorithmus von Bellman und Ford anstelle von Dijkstras Algorithmus verwendet werden, was zu einer Gesamtlaufzeit in $\mathcal{O}(|V|^2|E|)$ führt. Wir werden im Folgenden untersuchen, wie für einen Graphen mit allgemeinen Kantengewichten die kürzesten Pfade zwischen allen Paaren zweier Knoten schneller berechnet werden können.

Algorithmus von Floyd und Warshall Wir haben bereits in Abschnitt 5.1.6 einen Algorithmus zur Berechnung der transitiven Hülle eines Graphen $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ gesehen. Dieser berechnet eine Matrix $A = (a_{ij})$, in der ein Eintrag a_{ij} genau dann den Wert 1 hat, wenn es einen Weg vom Knoten v_i zum Knoten v_j gibt, und 0 sonst. Der Algorithmus iteriert über möglichen Zwischenknoten v_i und prüft für alle $u, v \in V$, ob es einen Weg von u nach v_i sowie einen Weg von v_i nach v gibt. In diesem Fall hätten wir auch einen Weg von u nach v gefunden.

Algorithmus von Floyd und Warshall

Der Algorithmus von Floyd und Warshall greift diese Idee wieder auf, nur dass diesmal die Länge eines kürzesten Weges zwischen zwei Knoten (statt eines binären Werts) verwaltet wird. Konkret sei d^i_{uv} mit $u, v \in V$ und $i \in \{0, \ldots, n\}$ die Länge eines kürzesten Weges von u nach v, auf dem alle Knoten v_k zwischen u und v einen Index $k \leq i$ haben.

Initialisierung

Initial ($i = 0$) sind keine Zwischenknoten erlaubt, also setzen wir $d^0_{uu} = 0$. Für $u \neq v$ setzen wir $d^0_{uv} = w((u, v))$, falls der Graph die Kante (u, v) enthält, und ansonsten $d^0_{uv} = \infty$ (da keine direkte Verbindung von u nach v existiert). Im Berechnungsschritt iterieren wir über alle $i \in \{1, 2, \ldots, n\}$, und für jedes i über alle $u \in V$ und alle $v \in V$. Wir setzen dann

Berechnungsschritt

$$d^i_{uv} = \min\left(d^{i-1}_{uv}, d^{i-1}_{uv_i} + d^{i-1}_{v_i v}\right), \tag{110}$$

denn entweder wird v_i auf einem kürzesten Weg nicht benutzt (dann ist $d^i_{uv} = d^{i-1}_{uv}$), oder die Benutzung von v_i führt zu einem kürzeren als dem bisher gefundenen uv-Weg (dann ist $d^i_{uv} = d^{i-1}_{uv_i} + d^{i-1}_{v_i v}$). Nach Ende der Berechnung enthält d^n_{uv} die Länge eines kürzesten Weges von u nach v, sofern ein solcher existiert. Wir müssen dann aber noch prüfen, ob es einen Kreis mit negativem Gewicht gibt. Dies ist genau dann der Fall, wenn es einen negativen Eintrag d^n_{uu} gibt.

Negative Kreise erkennen

Laufzeit

Der Algorithmus von Floyd und Warshall berechnet insgesamt $\mathcal{O}(|V|^3)$ viele Einträge d^i_{uv}, und jeder Eintrag kann in konstanter Zeit aus früher berechneten Einträgen berechnet werden. Die Zeit zur Prüfung, ob ein Kreis mit negativem Gewicht existiert, liegt in $\mathcal{O}(|V|)$. Damit berechnet der Algorithmus in Zeit $\mathcal{O}(|V|^3)$ die Distanzen zwischen allen Paaren zweier Knoten des Graphen. Man beachte, dass der Berechnungsschritt (110) in-place ausgeführt werden kann.

Rekonstruktion der Wege

Der obige Algorithmus berechnet nur die Abstände zwischen zwei Knoten, nicht die kürzesten Wege selbst. Dies kann aber leicht ergänzt werden, indem zu jedem Paar zweier Knoten (u, v) der Nachfolger $\text{succ}[u, v]$ von u auf einem kürzesten Weg von u nach v gespeichert wird. Initial setzen wir $\text{succ}[u, v] \leftarrow v$ für alle $u, v \in V$ mit $(u, v) \in E$, und $\text{succ}[u, v] \leftarrow \textbf{null}$ für alle $u, v \in V$ mit $(u, v) \notin E$. Wird dann im Berechnungsschritt $d^i_{uv} \leftarrow d^{i-1}_{uv_i} + d^{i-1}_{v_i v}$ gesetzt, muss $\text{succ}[u, v] \leftarrow \text{succ}[u, v_i]$ gesetzt werden. Um den kürzesten Weg von einem Knoten s zu einem Knoten t zu berechnen, setzen wir initial $u \leftarrow s$. Dann gibt $\text{succ}[u, t]$ den nächsten Knoten auf dem Weg zu t an. Wir setzen $u \leftarrow \text{succ}[u, t]$ und fahren auf die gleiche Art fort, bis t erreicht wird

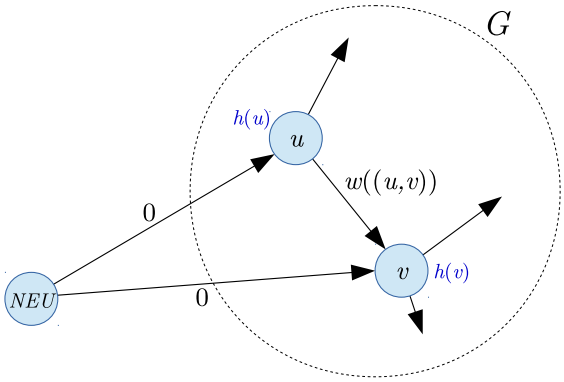


Abb. 5.15 Konstruktion des neuen Graphen.

(d.h., bis $u = t$ gilt). Die Zeit zur Rekonstruktion des Weges ist dann nur linear in der Weglänge, und auch die asymptotische Laufzeit des Algorithmus bleibt gleich.

Algorithmus von Johnson Der Algorithmus von Floyd und Warshall berechnet die kürzesten Wege zwischen allen Paaren zweier Knoten in Zeit $\mathcal{O}(|V|^3)$ und funktioniert auch dann, wenn Kanten mit negativem Gewicht existieren. Für Graphen, deren Kantengewichte stets nicht-negativ sind, können alle kürzesten Wege durch $|V|$-malige Ausführung von Dijkstras Algorithmus berechnet werden, was insgesamt Zeit $\mathcal{O}(|V|^2 \log |V| + |V||E|)$ kostet und damit im besten Fall schneller als der Algorithmus von Floyd und Warshall ist (und niemals schlechter). Wir werden jetzt den Algorithmus von Johnson beschreiben, der genau die gleiche Laufzeit hat, aber auch dann noch funktioniert, wenn es Kanten mit negativem Gewicht gibt.

VORBEREITUNG

Dazu fügen wir zunächst einen neuen NEU in V ein und erzeugen $|V|-1$ Kanten von NEU zu jedem $v \in V \backslash \{NEU\}$. Das Gewicht dieser Kanten ist 0. Zusätzlich führen wir für jeden Knoten $v \in V$ eine *Höhe* $h(v)$ ein und definieren eine neue Kantengewichtsfunktion $\hat{w}((u,v)) := w((u,v)) + h(u) - h(v)$. Das Ziel ist es nun, die Höhen $h(v)$ so zu wählen, dass die Kantengewichtsfunktion $\hat{w}$ stets nicht-negativ ist.

NUTZEN DER HÖHEN

Unabhängig von der Wahl der Höhen $h(v)$ beobachten wir, dass in dem Graphen mit den neuen Kantengewichten $\hat{w}((u,v))$ folgende zwei Eigenschaften gelten:

- Ein kürzester st-Pfad $P = \langle s, u_1, \ldots, u_k, t \rangle$ in $G = (V, E, w)$ ist auch ein kürzester st-Pfad in $G = (V, E, \hat{w})$ (also in Bezug auf die neuen Kantengewichte). Es gilt nämlich $\hat{w}(P) = w(P) + h(s) - h(t)$ (wobei wir vereinfachend $\hat{w}(P)$ und $w(P)$ schreiben, um die Gewichte von P bezüglich $\hat{w}$ bzw. w anzugeben), da sich die Höhen der Zwischenknoten gegeneinander aufheben. Der Term $\hat{w}(P)$ hängt also nur von den Höhen des Start- und Zielknotens ab, und diese sind für alle st-Wege gleich.

- Die Kosten eines Kreises $K = \langle u_1, u_2, \ldots, u_k, u_1 \rangle$ bleibt bezüglich w und $\hat{w}$ gleich, es gilt also $\hat{w}(K) = w(K)$.

WAHL DER HÖHEN

Als Höhe eines Knotens v wählen wir jetzt die Länge eines kürzesten Pfades vom neu eingefügten Knoten NEU zu v. Dann nämlich ist $h(v) \leq h(u) + w((u,v))$ (siehe Abbildung 5.15), und es gilt $\hat{w}((u,v)) = w((u,v)) + h(u) - h(v) \geq w((u,v)) + h(u) - (h(u) + w((u,v))) = 0$. Folglich sind alle Kantengewichte $\hat{w}((u,v))$ nicht-negativ.

Johnsons Algorithmus funktioniert also wie folgt: ALGORITHMUS

1) Füge dem Graphen einen neuen Knoten NEU sowie Kanten (NEU, v) zu allen anderen Knoten $v \in V \backslash \{NEU\}$ mit Gewicht 0 ein.

2) Benutze den Algorithmus von Bellman und Ford ausgehend von NEU zur Berechnung aller Höhen $h(v)$. Falls dabei ein Kreis mit negativem Gewicht gefunden wird, melde dies und brich ab.

3) Berechne die neuen Kantengewichte $\hat{w}((u,v)) = w((u,v)) + h(s) - h(t)$.

4) Für jeden Knoten $u \in V \backslash \{NEU\}$, starte Dijkstras Algorithmus von u aus, um die Wege zu allen Knoten $v \in V$ des Graphen zu berechnen. Danach müssen nur noch die gefundenen Distanzen um $h(u) - h(v)$ reduziert werden.

LAUFZEIT

Schritt 1 kostet Zeit $\mathcal{O}(|V|)$. Die Ausführung des Algorithmus von Bellman und Ford kostet Zeit $\mathcal{O}(|V||E|)$. Die neuen Kantengewichte $\hat{w}((u,v))$ können in Zeit $\mathcal{O}(|E|)$ berechnet werden. In Schritt 4 wird Dijkstras Algorithmus $|V| - 1$ Mal ausgeführt, was insgesamt Zeit $\mathcal{O}(|V|^2 \log |V| + |V||E|)$ dauert. Dies ist auch der dominierende Term in der Gesamtlaufzeit von Johnsons Algorithmus. Falls $|E| \in o(|V|^2)$ gilt, ist er damit also schneller als der Algorithmus von Floyd und Warshall.