Algorithmen
in Java

Von
Hans Werner Lang

2. Auflage

Oldenbourg Verlag München Wien

Prof. Dr. Hans Werner Lang ist seit 1994 Professor für Informatik an der Fachhochschule Flensburg.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Lektorat: Margit Roth
Herstellung: Anna Grosser
Umschlagkonzeption: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: Grafik + Druck, München
Bindung: Thomas Buchbinderei GmbH, Augsburg

ISBN 3-486-57938-X
ISBN 978-3-486-57938-3

Inhaltsverzeichnis

Vorwort

Algorithmen, also Rechenverfahren, sind für mich das faszinierendste Gebiet der Informatik. In einem der ersten Bücher über Computer, das ich als Schüler las, war beschrieben, dass der Computer die Quadratwurzel anders berechnet, als ich es in der Schule gelernt hatte. Hier wurde mir überhaupt zum erstenmal klar, dass es für ein mathematisches Problem nicht nur ein Lösungsverfahren, sondern mehrere und sogar höchst unterschiedliche Lösungsverfahren geben kann.

Eigentlich ist es ja nicht notwendig, noch ein neues Sortierverfahren zu erfinden, wenn es schon ein Sortierverfahren gibt. Oder doch?

Nun, es könnte schneller sein als andere Verfahren, es könnte mit weniger Speicherplatz auskommen, es könnte einfacher zu implementieren sein, es könnte leichter zu verstehen sein als andere Verfahren. D.h. es könnte von geringerer Komplexität sein als andere Verfahren, und zwar von geringerer Zeitkomplexität, Platzkomplexität oder Beschreibungskomplexität.

Das wichtigste Kriterium ist die Zeitkomplexität; wir werden alle Algorithmen ausführlich hinsichtlich ihrer Zeitkomplexität analysieren.

Oder das neue Verfahren könnte sich besser für kleine Datenmengen eignen, für teilweise vorsortierte Daten oder für Daten, in denen viele gleiche Elemente vorkommen. D.h. es könnte an spezielle Problemfälle besser angepasst sein als andere Verfahren.

Die unterschiedlichen Verfahren stehen durchaus in Konkurrenz zueinander, aber sie haben alle ihre Berechtigung, und wenn es allein wegen der in ihnen enthaltenen Ideen ist.

Dieses Buch stellt vielleicht mehr als andere die unterschiedlichen Verfahren zur Lösung jeweils des gleichen Problems gegenüber. Hierbei wird deutlich, dass man ein Problem mit einer Fülle von verschiedenen Ideen angehen kann. Dies sei auch zur Ermutigung gesagt, ruhig noch einmal neu über ein Problem nachzudenken, auch wenn es „schon gelöst“ ist.

Mir ist es bei der Beschäftigung mit String-Matching-Algorithmen passiert, dass ich mehr oder weniger nebenbei ein neues Verfahren, den Skip-Search-Algorithmus, gefunden habe. Allerdings stellte sich hinterher heraus, dass es das Verfahren im Prinzip doch schon gab, allerdings erst seit 1998.

Nicht nur die Idee kann mich an einem Algorithmus begeistern, sondern auch die Eleganz der Formulierung als Programm. Wenn es zwei unterschiedliche Versionen eines Algorithmus gibt, wobei die eine um 10% schneller ist, die andere aber eleganter,

entscheide ich mich immer für die zweite Version. Dies drückt sich auch an den in diesem Buch wiedergegebenen Programmen aus. Hin und wieder kann ein Programm mit ein paar Tricks vielleicht noch ein wenig schneller gemacht werden. Dann aber büßt es seine Eleganz ein, und das ist es nicht wert.

Wohlgemerkt gilt dies nur, wenn die erzielbare Beschleunigung sich im Bereich von weniger als einem Faktor 2 bewegt. Beim Sortierverfahren Heapsort etwa gibt es eine etwas kompliziertere Variante, die jedoch das Verfahren etwa doppelt so schnell macht, so dass es gegenüber Quicksort bestehen kann. In solch einem Fall ist ein wenig mehr an Beschreibungskomplexität in Kauf zu nehmen.

Dieses Buch ist ein Lehrbuch über Algorithmen für Studierende der Informatik an Fachhochschulen und Universitäten. Vorausgesetzt werden ein paar Kenntnisse in Mathematik, die Fähigkeit, in einer höheren Programmiersprache zu programmieren und Lust, sich mit Algorithmen zu beschäftigen.

Die Grundbegriffe der Mathematik und der Informatik, die in diesem Buch verwendet werden, wie Menge, Relation, Abbildung, Folge, Graph, Baum, Zeichenreihe, Sprache, Grammatik werden im Anhang behandelt. Wer sich also unsicher ist, was z.B. eine Abbildung ist, sollte zuerst den Anhang lesen. Gelegentlich wird auch auf den Anhang verwiesen, wenn eine bestimmte Definition für das Verständnis notwendig ist. Ich habe die Grundlagen in den Anhang verbannt, damit es in Kapitel 1 gleich mit Algorithmen losgeht.

An einigen wenigen Stellen wird man in diesem Buch Seiten finden, so z.B. im Abschnitt über das Sortierverfahren Shellsort, die sehr mathematisch aussehen, da sie aus einer scheinbar nicht enden wollenden Abfolge von „Definition, Satz, Beweis“ bestehen. Ich habe mich für diese Darstellung um der Klarheit willen entschieden. Es soll klar sein: hier wird etwas definiert, hier wird etwas behauptet, hier wird etwas bewiesen. Es handelt sich im übrigen stets um sehr einfach, Schritt für Schritt nachzuvollziehende Sachverhalte. Und es gesellt sich zum besseren Verständnis als Viertes auch immer noch das „Beispiel“ hinzu.

In vielen Büchern werden Algorithmen und Datenstrukturen zusammen behandelt, da manche Algorithmen auf speziellen Datenstrukturen basieren. Ich habe mich in diesem Buch auf Algorithmen konzentriert. Hierbei stehen die Algorithmen selbst und ihre Implementation in der Programmiersprache Java im Vordergrund; allgemeine Entwurfsprinzipien und Analysetechniken werden hierdurch nebenbei verdeutlicht.

Die ersten Kapitel behandeln Standardalgorithmen für das Sortieren, für die Textsuche, für Graphenprobleme, ferner aus dem Bereich der Codierung, der Kryptografie und der Rechnerarithmetik.

Etwas speziellere Themen sind Parser und Übersetzer (Kapitel 4), Sortiernetze (Kapitel 13) und parallele Sortieralgorithmen für Prozessorfelder (Kapitel 14).

Natürlich bedeutet ein Lehrbuch in erster Linie Arbeit und erst in zweiter Linie Spaß, aber vielleicht lässt sich das eine mit dem anderen vereinbaren, und so wünsche ich allen Lesern viel Freude beim Lesen und Lernen.

Es mögen Fehler und Unklarheiten in dem Buch enthalten sein; ich bitte in diesem Fall um eine kurze Nachricht unter lang@fh-flensburg.de. Eine Liste mit Korrekturen stelle ich unter www.inf.fh-flensburg.de/lang/algorithmen ins Internet.

Flensburg, Mai 2002 Hans Werner Lang

Vorwort zur 2. Auflage

Getreu dem Konzept dieses Buches, vielfältige unterschiedliche Lösungen für ein Problem darzustellen, ist in der neuen Auflage ein Kapitel mit mehreren Algorithmen zur Berechnung der konvexen Hülle einer endlichen Punktemenge hinzugekommen. Damit wird zugleich ein erster Einblick in das interessante Gebiet der algorithmischen Geometrie geboten.

Ein weiteres neues Kapitel behandelt den Begriff der Transformation von Problemen. Durch Transformation eines Problems A in ein anderes Problem B lässt sich in vielen Fällen entweder A schnell lösen, wenn sich B schnell lösen lässt, oder zeigen, dass sich B nicht schnell lösen lässt, wenn sich A nicht schnell lösen lässt.

Anwendungen solcher Transformationen sind die Multiplikation von Polynomen mit Hilfe der Fouriertransformation oder die Angabe einer unteren Schranke für die Zeitkomplexität der Berechnung der konvexen Hülle.

Transformationen sind auch ein wesentlicher Bestandteil der Komplexitätstheorie, die in einem neuen Kapitel zu NP-vollständigen Problemen gestreift wird. Als Beispiel für ein NP-vollständiges, also wahrscheinlich nicht effizient lösbares Problem wird das Travelling-Salesman-Problem herangezogen, und es werden Annäherungsverfahren zur Lösung des Travelling-Salesman-Problems angegeben.

Etwas über das klassische Gebiet der Algorithmen hinaus geht das neue Kapitel über die formale Verifikation von Programmen. Da aber die korrekte Implementation genauso wichtig ist wie der korrekte Entwurf eines Algorithmus, soll in diesem Kapitel ein erster Einblick in die Möglichkeiten und Grenzen der formalen Verifikation gegeben werden.

Flensburg, Juni 2006 Hans Werner Lang

Notation

Es ist eine ungeklärte Streitfrage, ob die Elemente einer endlichen Menge oder Folge von 1 bis n oder von 0 bis $n-1$ indiziert werden sollten. In der Mathematik erscheint das erste natürlicher, im Kontext einer Programmiersprache wie Java erscheint das zweite natürlicher, da Arrays von 0 bis $n-1$ indiziert werden. Ich habe daher durchgängig die Indizierung von 0 bis $n-1$ gewählt.

Die Menge der natürlichen Zahlen $\mathbb{N}$ beginnt in diesem Buch einheitlich bei 1. Soll die Null mit dazugehören, so wird das Zeichen $\mathbb{N}_0$ verwendet.

Ferner bezeichnet $\mathbb{Z}$ die ganzen Zahlen, $\mathbb{Q}$ die rationalen Zahlen, $\mathbb{R}$ die reellen Zahlen, $\mathbb{C}$ die komplexen Zahlen sowie $\mathbb{B}$ die boolesche Menge $\{0, 1\}$.

Gelegentlich treten in mathematischen Aussagen die Zeichen $\forall$ und $\exists$ auf; sie bedeuten „für alle“ bzw. „es gibt“.

Für die ganzzahlige Division mit Rest werden die Operationssymbole div und mod benutzt. Beispielsweise ergibt 17 div 5 den Quotienten 3, 17 mod 5 den Rest 2.

Ich habe mich bemüht, Formeln möglichst einfach zu halten. So habe ich weitgehend auf Indizes verzichtet, wenn aus dem Zusammenhang klar wird, was gemeint ist. So heißt es beispielsweise: Quicksort hat im Durchschnitt eine Zeitkomplexität von

$$T(n) \in \Theta(n \log(n))$$

anstelle von

$$T^{avg}_{Quick}(n) \in \Theta(n \log(n))$$

Danksagung

Ich danke meinem Freund und Kollegen Manfred Schimmler sowie Viktor Bunimov für die sorgfältige Durchsicht einer ersten Version des Manuskripts und für die damit verbundenen Anregungen und Hinweise.

1 Algorithmen und Komplexität

Algorithmen kennen wir aus dem täglichen Leben. Ein *Algorithmus* ist eine Schritt-für-Schritt-Anleitung für einen bestimmten Vorgang.

Beispiele sind etwa eine Wegbeschreibung:

> „*Fahren Sie geradeaus bis zur nächsten Ampel. Dort fahren Sie links und dann solange geradeaus, bis Sie an einen Kreisel kommen.*“

oder eine Gebrauchsanweisung:

> „*Innensechskantschraube mit Sechskantstiftschlüssel lösen und herausnehmen. Spannflansch abnehmen.*“

oder ein Kochrezept:

> „*Die Milch zum Kochen bringen. Das aufgelöste Puddingpulver hineingeben und glatt rühren. Kurz aufkochen und erkalten lassen.*“

Ein wichtiges Kennzeichen eines Algorithmus ist, dass die Schritte so eindeutig und detailliert angegeben sind, dass sie von demjenigen, der den Algorithmus ausführen soll, unmittelbar umgesetzt werden können. Bei dem Kochrezept ist also vorausgesetzt, dass der Koch den Begriff „Rühren“ kennt und ihn umsetzen kann.

In vielen Fällen enthalten Algorithmen auch sogenannte Schleifen: ein Vorgang wird so lange wiederholt, bis eine bestimmte Bedingung eintritt. „Glatt rühren“ bedeutet: Rühren, prüfen ob die Masse glatt ist, wenn nicht, weiterrühren, erneut prüfen usw.

Wichtig ist, dass eine solche Schleife nicht unendlich lange wiederholt wird, sondern dass sie irgendwann abbricht. Wir verlangen von einem Algorithmus, dass er nach endlich vielen Schritten zu einem Ende kommt.

Gelegentlich treffen wir in Algorithmen auch auf komplexere Anweisungen, die nicht unmittelbar umgesetzt werden können:

> „*Bereiten Sie einen Hefeteig zu.*“
> „*Installieren Sie zunächst den XY-Treiber.*“

Hier handelt es sich um Aufrufe von Unter-Algorithmen, die an anderer Stelle detailliert angegeben sind, etwa unter „Grundrezepte“ bzw. „Treiberinstallation“. In Algorithmen dürfen jedoch keine ungenauen Anweisungen vorkommen wie

> „*Treffen Sie alle erforderlichen Vorbereitungen.*“

Maschinenmodell

Wenden wir uns nun den Computer-Algorithmen zu. Um Computer-Algorithmen angeben zu können, ist es notwendig zu wissen, welche Schritte der Computer unmittelbar umsetzen kann. Um größtmögliche Allgemeinheit herzustellen, wird der Computer dargestellt durch ein *abstraktes Maschinenmodell*. Das Maschinenmodell, das den heutzutage im Gebrauch befindlichen Computern entspricht, ist die *Random-Access-Maschine*.

Die Random-Access-Maschine arbeitet mit Maschinenwörtern fester Länge (z.B. 32 Bit), die Zahlen oder Zeichen darstellen. Sie speichert die Maschinenwörter in einem unbeschränkt großen Speicher. Sie kann in einem Schritt ein Maschinenwort in den Speicher schreiben oder aus dem Speicher lesen (daher *random access* – wahlfreier Zugriff auf den Speicher). Ebenfalls in einem Schritt kann sie eine Rechenoperation mit zwei Maschinenwörtern, also eine Addition, Subtraktion, Multiplikation, Division oder einen Vergleich ausführen.

Es gibt aber auch noch andere Maschinenmodelle. Für parallele Algorithmen gibt es eine Fülle von möglichen Maschinenmodellen. Wir werden das zweidimensionale Prozessorfeld als eines der am leichtesten zu realisierenden Modelle benutzen.

Das einfachste, aber gleichwohl universelle abstrakte Maschinenmodell ist die Turingmaschine. Mit einer Turingmaschine lässt sich alles berechnen, was berechenbar ist.

Darüber hinaus gibt es spezielle abstrakte Maschinenmodelle wie z.B. nichtdeterministische Maschinen.

Zeitkomplexität

Wenn wir wissen, welche Schritte die Maschine umsetzen kann, können wir Algorithmen für die Maschine formulieren, und wir können zählen, wie viele Schritte die Maschine benötigt, um den Algorithmus auszuführen.

Die Anzahl der Schritte wird im allgemeinen von der Eingabe abhängen. Ein Sortieralgorithmus wird zum Sortieren von 1000 Zahlen länger brauchen als zum Sortieren von 10 Zahlen. Und er wird möglicherweise zum Sortieren von 10 Zahlen länger brauchen, wenn die Zahlen in umgekehrter Reihenfolge eingegeben werden, als wenn sie in schon sortierter Reihenfolge eingegeben werden.

Zum einen spielt also die Länge der Eingabe, d.h. die Größe des Problems, das der Algorithmus lösen soll, eine Rolle. Zum anderen ist auch die Eingabe selbst von Bedeutung.

Die Anzahl der Schritte, die ein Algorithmus benötigt, wird daher immer als Funktion $T(n)$ in Abhängigkeit von der Problemgröße n angegeben. Die Funktion $T(n)$ heißt *Zeitkomplexität* des Algorithmus, oder kurz *Komplexität* des Algorithmus, wenn klar ist, dass wir vom Zeitverhalten reden.

Um von der jeweiligen konkreten Eingabe unabhängig zu sein, gibt man für $T(n)$ folgendes an:

- die maximale Anzahl von Schritten, die bei einer beliebigen Eingabe der Länge n benötigt wird (Zeitkomplexität im schlechtesten Fall – *worst case*) oder
- die durchschnittliche Anzahl von Schritten, die bei einer beliebigen Eingabe der Länge n benötigt wird (Zeitkomplexität im Durchschnitt – *average case*).

Die Zeitkomplexität wird meistens in der O-Notation angegeben. Hierbei bedeutet z.B.

$$T(n) \in O(n^2),$$

dass der Algorithmus höchstens proportional zu n^2 viele Schritte benötigt, um ein Problem der Größe n zu lösen. In ähnlicher Weise bedeutet

$$T(n) \in \Omega(n^2),$$

dass der Algorithmus mindestens proportional zu n^2 viele Schritte benötigt, um ein Problem der Größe n zu lösen. Schließlich bedeutet

$$T(n) \in \Theta(n^2),$$

dass der Algorithmus mindestens, aber auch höchstens proportional zu n^2 viele Schritte benötigt, um ein Problem der Größe n zu lösen.

Die genaue Definition der O-Notation ist im Anhang angegeben.

Neben der Korrektheit eines Algorithmus spielt seine Zeitkomplexität die wichtigste Rolle. Sicherlich ist ein Algorithmus nicht zu gebrauchen, wenn er nicht korrekt ist, d.h. wenn er nicht in allen Fällen das Ergebnis liefert, das er liefern soll. Aber auch ein Algorithmus mit einer Zeitkomplexität von $T(n) \in \Theta(2^n)$ ist nicht zu gebrauchen, höchstens für kleine Problemgrößen wie etwa $n = 30$. Schon bei einer Problemgröße von $n = 100$ ergäbe sich eine Rechenzeit, die größer ist als das Alter des Universums. Und auch nächstes Jahr, obwohl das Universum dann älter ist und es doppelt so schnelle Rechner gibt, gilt dasselbe immer noch für $n = 101$.

Effiziente Algorithmen

Eine Komplexitätsfunktion, bei der n im Exponenten auftritt, wird als *exponentielle Komplexität* bezeichnet. Leider sind für manche Probleme, insbesondere Optimierungsprobleme, nur Algorithmen mit exponentieller Komplexität bekannt. Das bekannteste Beispiel ist das Travelling-Salesman-Problem (TSP). Ein Handlungsreisender soll auf seiner Rundreise durch n Städte möglichst wenig Kilometer zurücklegen. In welcher Reihenfolge muss er die Städte besuchen?

Ein Algorithmus für die Lösung dieses Problems ist schnell geschrieben: alle Rundreisen der Reihe nach erzeugen, jeweils deren Längen berechnen und hiervon am Ende das Minimum bilden. Allerdings hat dieser Algorithmus mindestens exponentielle Komplexität, denn es gibt $(n-1)!$ verschiedene Rundreisen durch n Städte.[1]

[1] Für die Fakultätsfunktion $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ gilt offenbar $n! \geq (n/2)^{n/2}$

Es ist bis heute kein Algorithmus mit polynomieller Komplexität für dieses Problem und für eine ganze Reihe ähnlicher Probleme bekannt, und es ist zweifelhaft, ob überhaupt ein solcher Algorithmus existieren kann. *Polynomielle Komplexität* bedeutet, dass $T(n) \in O(n^k)$, wobei k eine Konstante ist.

Ein Algorithmus mit polynomieller Komplexität wird auch ein *effizienter Algorithmus* genannt. Zwar ist ein Algorithmus mit einer Komplexität von $T(n) \in O(n^{50})$ nicht wirklich effizient; in der Praxis aber treten meist nur Algorithmen bis hin zu $O(n^4)$ auf.

Wirklich schnell sind allerdings nur die Algorithmen bis hin zu einer Komplexität von $T(n) \in O(n \log(n))$.

Wir werden dies im nächsten Kapitel am Beispiel der Sortieralgorithmen sehen. Die einfachen Sortieralgorithmen mit der Komplexität von $T(n) \in O(n^2)$ sind schon kaum noch zu gebrauchen, wenn es etwa darum geht, eine Million Einträge eines Telefonbuches zu sortieren. Denn die Anzahl der Schritte bewegt sich im Bereich von Billionen. Bei den schnellen Sortieralgorithmen mit ihrer Komplexität von $T(n) \in O(n \log(n))$ beträgt die Anzahl der Schritte dagegen lediglich einige Zig-Millionen.

2 Sortieren

Das Sortieren einer Datenfolge ist eines der am leichtesten zu verstehenden und am häufigsten auftretenden algorithmischen Probleme.

In seiner einfachsten Form besteht das Problem darin, eine endliche Folge von ganzen Zahlen so umzuordnen, dass die Zahlen in aufsteigender Reihenfolge auftreten. In genau dieser einfachsten Form werden wir das Problem behandeln; alle in Kapitel 2 vorgestellten Sortieralgorithmen beziehen sich auf diese Form des Sortierproblems. Die den Algorithmen zugrunde liegenden Ideen werden so am deutlichsten sichtbar.

Anstelle von Zahlen kann man natürlich auch andere Daten sortieren, z.B. Zeichenreihen. Voraussetzung ist, dass die Daten untereinander vergleichbar sind, d.h. dass eine Ordnungsrelation $\leq$ auf der Menge der Daten definiert ist.

In vielen Fällen sind die Daten ganze Datensätze, die nach einem bestimmten Kriterium sortiert werden sollen. Beispielsweise könnten dies Daten von Personen sein, die nach dem Geburtsjahr sortiert werden sollen.

Zur Behandlung dieser etwas weiter gefassten Fälle des Sortierproblems müssen die Algorithmen entsprechend angepasst werden.

Es folgt eine formale Definition des Sortierproblems. Die Begriffe wie endliche Folge, Abbildung oder Permutation sind im Anhang erklärt.

Definition: Sei $n \in \mathbb{N}$ und $a = a_0, ..., a_{n-1}$ eine endliche Folge mit $a_i \in \mathbb{N}$ $(i = 0, ..., n-1)$.

Das *Sortierproblem* besteht darin, eine Folge $a_{\varphi(0)}, ..., a_{\varphi(n-1)}$ zu finden, derart dass $a_{\varphi(i)} \leq a_{\varphi(j)}$ ist für alle $i, j \in \{0, ..., n-1\}$ mit $i < j$ und derart dass die Abbildung φ eine Permutation der Indexmenge $\{0, ..., n-1\}$ ist.

Beispiel: Es seien $n = 8$ und $a = 3\ 8\ 1\ 4\ 3\ 3\ 2\ 6$.

i :	0 1 2 3 4 5 6 7
a_i :	3 8 1 4 3 3 2 6
$\varphi(i)$:	2 6 5 0 4 3 7 1
$a_{\varphi(i)}$:	1 2 3 3 3 4 6 8

2.1 Insertionsort

Insertionsort (Sortieren durch Einfügen) ist ein elementares Sortierverfahren. Es hat eine Zeitkomplexität von $\Theta(n^2)$, ist damit also langsamer als etwa Heapsort, Mergesort oder auch Shellsort. Sehr gut geeignet ist Insertionsort für das Sortieren von kleinen Datenmengen oder für das Einfügen von weiteren Elementen in eine schon sortierte Folge.

Idee

Zu Anfang und nach jedem Schritt des Verfahrens besteht die zu sortierende Folge a_0, ..., a_{n-1} aus zwei Teilen: Der erste Teil a_0, ..., a_{i-1} ist bereits aufsteigend sortiert, der zweite Teil a_i, ..., a_{n-1} ist noch unsortiert.

Das Element a_i wird als nächstes in den bereits sortierten Teil eingefügt, indem es der Reihe nach mit a_{i-1},a_{i-2} usw. verglichen wird. Sobald ein Element a_j mit $a_j \leq a_i$ gefunden wird, wird a_i hinter diesem eingefügt. Wird kein solches Element gefunden, wird a_i an den Anfang der Folge gesetzt.

Damit ist der sortierte Teil um ein Element länger geworden. Im nächsten Schritt wird a_{i+1} in den sortierten Teil eingefügt usw. Zu Anfang besteht der sortierte Teil nur aus dem Element a_0; zum Schluss aus allen Elementen a_0, ..., a_{n-1}.

Beispiel: Die folgende Tabelle zeigt die Sortierschritte zum Sortieren der Folge 5 7 0 3 4 2 6 1. Der jeweils bereits sortierte Teil der Folge ist grau hinterlegt dargestellt. Ganz rechts steht in Klammern die Anzahl der Positionen, um die das eingefügte Element nach links gewandert ist.

5	7	0	3	4	2	6	1	(0)
5	7	0	3	4	2	6	1	(0)
0	5	7	3	4	2	6	1	(2)
0	3	5	7	4	2	6	1	(2)
0	3	4	5	7	2	6	1	(2)
0	2	3	4	5	7	6	1	(4)
0	2	3	4	5	6	7	1	(1)
0	1	2	3	4	5	6	7	(6)

Implementierung

Die folgende Funktion *insertionsort* sortiert ein Integer-Array $a[0]$, ..., $a[n-1]$.

Die Sortierfunktion ist in der Klasse *InsertionSorter* gekapselt. Mit den Anweisungen

```
InsertionSorter s=new InsertionSorter();
s.sort(b);
```

wird ein Objekt vom Typ *InsertionSorter* erzeugt und anschließend die Methode *sort* zum Sortieren eines Arrays b aufgerufen.

```
public class InsertionSorter
{
    private int[] a;
    private int n;

    public void sort(int[] a)
    {
        this.a=a;
        n=a.length;
        insertionsort();
    }

    private void insertionsort()
    {
        int i, j, t;
        for (i=1; i<n; i++)
        {
            j=i;
            t=a[j];
            while (j>0 && a[j-1]>t)
            {
                a[j]=a[j-1];
                j--;
            }
            a[j]=t;
        }
    }

}    // end class InsertionSorter
```

Analyse

Im schlechtesten Fall wird der Platz für das einzufügende Element immer erst ganz am Anfang des sortierten Teils gefunden. D.h. in der While-Schleife werden Folgen der Länge $1, 2, 3, \ldots, n-1$ durchsucht. Insgesamt sind dies $(n-1) \cdot n/2$ Schritte, also $\Theta(n^2)$ Schritte. Dieser Fall tritt ein, wenn die Folge zu Anfang in absteigender Reihenfolge sortiert ist.

Es ginge auch schneller, die Einfügeposition des Elements a_i innerhalb des sortierten Teils a_0, ..., a_{i-1} zu finden, nämlich mit binärer Suche. Da aber die größeren Elemente alle nach rechts rücken müssen, um die Einfügeposition frei zu machen, ist für das Einfügen ohnehin lineare Zeit erforderlich.

Die genaue Anzahl der Schritte, die Insertionsort benötigt, wird durch die Anzahl der Inversionen der zu sortierenden Folge bestimmt.

Definition: Sei $a = a_0, ..., a_{n-1}$ eine endliche Folge. Eine *Inversion* ist ein Paar (i, j) mit $i < j$ und $a_i > a_j$. Eine Inversion ist also ein Paar von Indexpositionen, an denen die Elemente der Folge in falscher Reihenfolge stehen.[1]

Beispiel: Sei $a = 5, 7, 4, 9, 7$. Dann ist $(0, 2)$ eine Inversion, denn es ist $a_0 > a_2$, nämlich $5 > 4$. Außerdem sind $(1, 2)$ und $(3, 4)$ Inversionen, da $7 > 4$ und $9 > 7$. Weitere Inversionen sind nicht vorhanden.

Wir bestimmen nun die Anzahl der Inversionen (i, j) der Folge a getrennt für jede Position j. Ergebnis ist jeweils ein Wert v_j, der die Anzahl der Elemente a_i angibt, die links von a_j stehen und größer sind als a_j.

In der Folge $a = 5, 7, 4, 9, 7$ stehen beispielsweise links von $a_2 = 4$ die zwei größeren Zahlen 5 und 7, also ist $v_2 = 2$. Links von $a_4 = 7$ steht nur eine größere Zahl, also ist $v_4 = 1$.

Die Folge der v_j wird als Inversionenfolge bezeichnet.

Definition: Die *Inversionenfolge* $v = v_0, ..., v_{n-1}$ einer Folge $a = a_0, ..., a_{n-1}$ ist definiert durch

$$v_j = |\{(i, j) \mid i < j \ \wedge \ a_i > a_j\}|$$

für $j = 0, ..., n - 1$.

Beispiel: Die obige Folge $a = 5, 7, 4, 9, 7$ hat die Inversionenfolge $v = 0, 0, 2, 0, 1$.

Offensichtlich gilt $v_i \leq i$ für alle $i = 0, ..., n - 1$. Genau dann, wenn alle v_i gleich 0 sind, ist die zugehörige Folge a sortiert. Ist die Folge a eine Permutation, so ist sie durch ihre Inversionenfolge v eindeutig bestimmt. Die Permutation $n - 1, ..., 0$ hat die Inversionenfolge $0, ..., n - 1$.

Satz: Sei $a = a_0, ..., a_{n-1}$ eine Folge und $v = v_0, ..., v_{n-1}$ ihre Inversionenfolge. Dann ist die Anzahl der Schritte, die Insertionsort zum Sortieren der Folge benötigt

$$T(a) = \sum_{i=0,...,n-1} v_i$$

[1] Wenn die Folge a eine Permutation ist, lässt sich eine Inversion auch durch (a_i, a_j) anstelle von (i, j) angeben [Knu 73].

Beweis: Offensichtlich benötigt Insertionsort in jeder Iteration i gerade v_i Schritte, um das Element a_i einzufügen. Daher ist die Gesamtzahl der benötigten Schritte gleich der Summe aller v_i.

Beispiel: Die folgende Tabelle zeigt die Folge a aus dem Anfangsbeispiel und die zugehörige Inversionenfolge.

i	0	1	2	3	4	5	6	7
a_i	5	7	0	3	4	2	6	1
v_i	0	0	2	2	2	4	1	6

Beispielsweise ist $v_5 = 4$, weil vier Elemente links von $a_5 = 2$ stehen, die größer als 2 sind (nämlich 5, 7, 3 und 4). Entsprechend benötigt Insertionsort zum Einfügen der 2 genau 4 Schritte.

Die Summe aller v_i, also die Gesamtzahl aller Inversionen, ist 17. Insertionsort benötigt also zum Sortieren der Folge 17 Schritte.

2.2 Quicksort

Der *Quicksort*-Algorithmus [Hoa 62] ist eines der schnellsten und zugleich einfachsten Sortierverfahren. Das Verfahren arbeitet rekursiv nach dem *Divide-and-Conquer-Prinzip*.

Die Divide-and-Conquer-Strategie zur Lösung eines Problems besteht aus drei Schritten:

1) *Divide* das Problem wird in Teilprobleme zerlegt;
2) *Conquer* die Teilprobleme werden gelöst;
3) *Combine* die Lösungen der Teilprobleme werden zusammengefügt.

Diese Strategie funktioniert immer dann, wenn die Lösungen der Teilprobleme nach dem Zusammenfügen die Lösung des ursprünglichen Problems ergeben.

Divide and conquer bedeutet *teile und herrsche*.

Idee

Bild 2.1 zeigt schematisch die Vorgehensweise von Quicksort anhand einer Eingabefolge von Nullen (weiß) und Einsen (grau). Zunächst wird die zu sortierende Folge a so in zwei Teilstücke b und c zerlegt, dass alle Elemente des ersten Stücks b kleiner oder gleich allen Elementen des zweiten Stücks c sind (*divide*). Danach werden die beiden Teilstücke sortiert, und zwar rekursiv nach demselben Verfahren (*conquer*). Wieder zusammengesetzt ergeben die Teilstücke die sortierte Folge (*combine*).

Die Aufteilung geschieht, indem zunächst ein Vergleichselement x gewählt wird. Alle Elemente der Folge, die kleiner als x sind, kommen in das erste Teilstück. Alle Elemente,

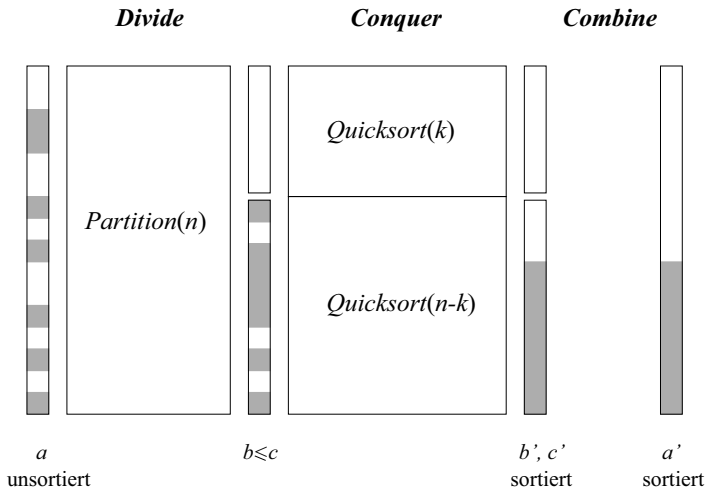

Bild 2.1: *Quicksort(n)*

die größer als x sind, kommen in das zweite Teilstück. Bei Elementen, die gleich x sind, ist es egal, in welches Teilstück sie kommen.

In dem folgenden Aufteilungsverfahren kann es auch vorkommen, dass ein Element, das gleich x ist, zwischen den beiden Teilstücken verbleibt. Dieses Element steht dann schon an seiner richtigen Position.

Algorithmus *Partition*

Eingabe: Folge a_0, ..., a_{n-1} mit n Elementen

Ausgabe: Umordnung der Folge derart, dass alle Elemente a_0, ..., a_j kleiner oder gleich den Elementen a_i, ..., a_{n-1} sind $(i > j)$

Methode: wähle das mittlere Element der Folge als Vergleichselement x;

setze $i = 0$ und $j = n - 1$;

wiederhole solange $i \leq j$

suche von links das erste Element a_i mit $a_i \geq x$;

suche von rechts das erste Element a_j mit $a_j \leq x$;

falls $i \leq j$

vertausche a_i und a_j;

setze $i = i + 1$ und $j = j - 1$;

Quicksort behandelt nach dieser Aufteilung der Folge die beiden Teilstücke a_0, ..., a_j und a_i, ..., a_{n-1} nach demselben Verfahren rekursiv weiter. Wenn ein im Verlauf des Verfahrens entstehendes Teilstück nur aus einem Element besteht, endet die Rekursion.

Programm

Die folgende Funktion *quicksort* sortiert ein Teilstück des Arrays a vom Index lo bis zum Index hi.

Die Sortierfunktion ist in der Klasse *QuickSorter* gekapselt. Die Methode *sort* übergibt das zu sortierende Array an das Array a, setzt n auf dessen Länge und ruft *quicksort* mit dem unteren Index 0 und dem oberen Index n-1 auf.

Die Hilfsfunktion *exchange*(i, j) vertauscht die Array-Elemente $a[i]$ und $a[j]$.

Mit den Anweisungen

```
QuickSorter s=new QuickSorter();
s.sort(b);
```

wird ein Objekt vom Typ *QuickSorter* erzeugt und anschließend die Methode *sort* zum Sortieren eines Arrays b aufgerufen. Es folgt das Programm.

```
public class QuickSorter
{
    private int[] a;
    private int n;


    public void sort(int[] a)
    {
        this.a=a;
        n=a.length;
        quicksort(0, n-1);
    }


    private void quicksort (int lo, int hi)
    {
        int i=lo, j=hi;
        int x=a[(lo+hi)/2];
```

```
        //  Aufteilung
        while (i<=j)
        {
            while (a[i]<x) i++;
            while (a[j]>x) j--;
            if (i<=j)
            {
                exchange(i, j);
                i++; j--;
            }
        }

        // Rekursion
        if (lo<j) quicksort(lo, j);
        if (i<hi) quicksort(i, hi);
    }

    private void exchange(int i, int j)
    {
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }

}    // end class QuickSorter
```

Analyse

Der Algorithmus verläuft optimal, wenn jeder Aufteilungsschritt im Verlauf der Rekursion jeweils etwa gleichlange Teilstücke erzeugt. In diesem günstigsten Fall benötigt Quicksort $\Theta(n \log(n))$ Schritte, denn die Rekursionstiefe ist $\log(n)$ und in jeder Schicht sind n Elemente zu behandeln (Bild 2.2 a).

Der ungünstigste Fall tritt ein, wenn ein Teilstück stets nur aus einem Element und das andere aus den restlichen Elementen besteht. Dann ist die Rekursionstiefe $n-1$ und Quicksort benötigt $\Theta(n^2)$ Schritte (Bild 2.2 c).

Im Mittel wird etwa eine Aufteilung wie in Bild 2.2 b dargestellt zustande kommen.

Die Wahl des Vergleichswertes x spielt die entscheidende Rolle dafür, welche Aufteilung zustande kommt. Man könnte z.B. auch das erste Element der Folge als Vergleichselement wählen. Dies würde aber dazu führen, dass das ungünstigste Verhalten des Algorithmus ausgerechnet dann eintritt, wenn die Folge zu Anfang bereits sortiert ist. Daher ist es besser, das mittlere Element der Folge zu wählen.

Am besten ist es natürlich, als Vergleichselement dasjenige Element zu wählen, das von der Größe her in der Mitte der Elemente liegt (der Median). Dann würde die optimale

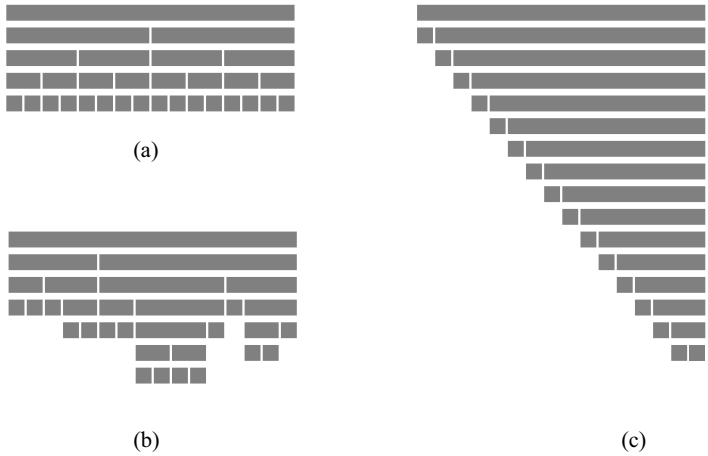

Bild 2.2: *Rekursionstiefe von Quicksort: (a) im besten Fall, (b) im Mittel, (c) im schlechtesten Fall*

Aufteilung zustande kommen. Tatsächlich ist es möglich, den Median in linearer Zeit zu bestimmen (Abschnitt 2.10). In dieser Variante würde Quicksort also auch im schlechtesten Fall nur $O(n \log(n))$ Schritte benötigen.

Quicksort lebt jedoch von seiner Einfachheit. Außerdem zeigt es sich, dass Quicksort auch in der einfachen Form im Durchschnitt nur $O(n \log(n))$ Schritte benötigt. Überdies ist die Konstante, die sich in der O-Notation verbirgt, sehr klein. Wir nehmen dafür in Kauf, dass Quicksort im (seltenen) schlechtesten Fall $\Theta(n^2)$ Schritte benötigt.

Satz: Die Zeitkomplexität von Quicksort beträgt

$\Theta(n \log(n))$	im Durchschnitt und
$\Theta(n^2)$	im schlechtesten Fall.

Zusammenfassung

Quicksort erweist sich in der Praxis als das schnellste Sortierverfahren. Es hat eine Zeitkomplexität von $\Theta(n \log(n))$ im Durchschnitt. Im (seltenen) schlechtesten Fall ist es mit einer Zeitkomplexität von $\Theta(n^2)$ allerdings genauso langsam wie Insertionsort.

Es gibt Sortierverfahren, die auch im schlechtesten Fall in $O(n \log(n))$ liegen, z.B. Heapsort und Mergesort. Diese sind jedoch im Durchschnitt um einen konstanten Faktor langsamer als Quicksort.

Es ist möglich, mit einer Variante von Quicksort auch im schlechtesten Fall eine Zeitkomplexität von $O(n \log(n))$ zu erreichen (indem als Vergleichselement der Median

gewählt wird). Dieses Verfahren ist jedoch im Durchschnitt und im schlechtesten Fall um einen konstanten Faktor langsamer als Heapsort oder Mergesort; daher ist es für die Praxis nicht interessant.

2.3 Heapsort

Das Sortierverfahren *Heapsort* [Wil 64] hat eine Zeitkomplexität von $\Theta(n \log(n))$. Eine untere Schranke für die Zeitkomplexität von Sortierverfahren ist $\Omega(n \log(n))$. Heapsort ist daher *optimal*, d.h. es gibt kein asymptotisch schnelleres Sortierverfahren (siehe Abschnitt 2.8).

Heapsort verwendet eine besondere Datenstruktur, die als *Heap* bezeichnet wird.[2] Diese Datenstruktur wird im Folgenden erklärt; sie basiert auf der im Anhang (Grundlagen) angegebenen Definition eines (fast) vollständigen binären Baums. Ein binärer Baum ist (fast) vollständig, wenn alle Schichten außer möglicherweise der letzten vollständig besetzt sind.

Grundlagen

Definition: Sei $T = (V, E)$ ein (fast) vollständiger binärer Baum mit einer Knotenmarkierung $a : V \to M$, die jedem Knoten u eine Markierung $a(u)$ aus einer geordneten Menge $(M, \leq)$ zuordnet.

Ein Knoten $u \in V$ hat die *Heap-Eigenschaft*, wenn er keinen direkten Nachfolgerknoten mit einer größeren Markierung hat, oder anders ausgedrückt, wenn gilt:

$$\forall\, v \in V \;:\; (u, v) \in E \quad \Rightarrow \quad a(u) \geq a(v)$$

T heißt *Heap*, wenn alle Knoten die Heap-Eigenschaft haben, d.h. wenn gilt:

$$\forall\, (u, v) \in E \;:\; a(u) \geq a(v)$$

Wir nennen T einen *Semi-Heap*, wenn alle Knoten außer möglicherweise der Wurzel r des Baumes die Heap-Eigenschaft haben, d.h. wenn gilt:

$$\forall\, (u, v) \in E,\; u \neq r \;:\; a(u) \geq a(v)$$

Beispiel: In Bild 2.3 ist ein Heap mit 10 Knoten dargestellt.

Alle Blätter haben automatisch die Heap-Eigenschaft, da sie keine Nachfolgerknoten haben, somit insbesondere keine mit einer größeren Markierung.

[2] Die Bezeichnung *Heap* wird auch für den Speicherplatz verwendet, den ein Computer für dynamisch erzeugte Variablen benutzt. Die beiden Begriffe haben jedoch inhaltlich nichts miteinander zu tun.

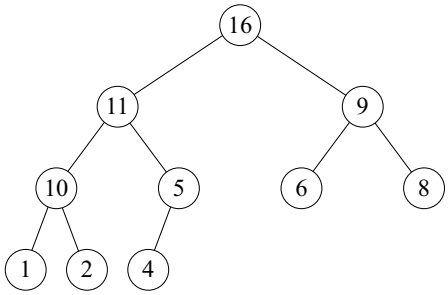

Bild 2.3: *Heap mit $n = 10$ Knoten*

Sortierverfahren

Die Datenstruktur des Heapsort-Verfahrens ist ein binärer Baum, dessen Knoten die zu sortierenden Daten enthalten. In der Implementation wird dieser Baum später in einem Array gespeichert. Es ist somit nicht nötig, den Baum als Zeiger-Struktur zu realisieren.

Heapsort-Algorithmus

Die folgende Beschreibung des Heapsort-Algorithmus nimmt Bezug auf Bild 2.4 a–e.

Wenn die zu sortierenden Daten als Heap arrangiert sind, lässt sich das größte Datenelement unmittelbar an der Wurzel des Baumes entnehmen und ausgeben (a). Um das nächstgrößte Element zu entnehmen, müssen die verbleibenden Elemente zunächst als Heap neu angeordnet werden.

Dies geschieht in folgender Weise: Sei b ein Blatt maximaler Tiefe. Die Markierung von b wird an die Stelle der Wurzel geschrieben; anschließend wird b gelöscht (b). Das Ergebnis ist ein Semi-Heap, denn die Wurzel hat nun möglicherweise nicht mehr die Heap-Eigenschaft.

Aus einem Semi-Heap einen Heap zu machen ist einfach: Wenn die Wurzel bereits die Heap-Eigenschaft hat, ist gar nichts zu tun. Wenn nicht, wird ihre Markierung mit der Markierung eines ihrer direkten Nachfolger vertauscht, und zwar mit der maximalen (c). Damit hat nun die Wurzel die Heap-Eigenschaft, aber dafür möglicherweise der entsprechende Nachfolger v nicht mehr. Es wird nun mit v fortgefahren, d.h. der Semi-Heap mit Wurzel v wird zum Heap gemacht (d). Das Verfahren endet spätestens an einem Blatt, da Blätter stets die Heap-Eigenschaft haben (e).

Nachdem nun die verbleibenden Elemente wieder als Heap angeordnet sind, kann das nächstgrößte Element entnommen werden usw. Die Datenelemente werden also in absteigend sortierter Reihenfolge ausgegeben.

Das Umarrangieren eines Semi-Heaps zu einem Heap lässt sich konzeptuell mit folgender Prozedur *downheap* bewirken:

Prozedur *downheap* (v)

Eingabe: Semi-Heap mit Wurzel v

Ausgabe: Heap (durch Umordnung der Knotenmarken)

Methode: wiederhole solange v nicht die Heap-Eigenschaft hat

wähle Nachfolgerknoten w mit maximaler Markierung $a(w)$;

vertausche $a(v)$ und $a(w)$;

setze $v = w$;

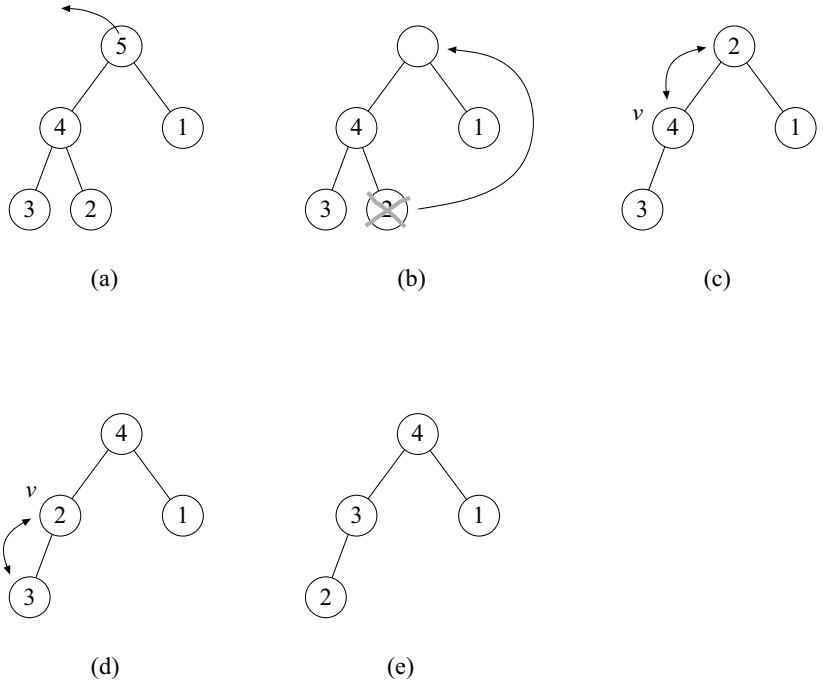

Bild 2.4: *Entnehmen des maximalen Elementes und Neuordnen des Heaps*

Die Prozedur *downheap* wird auch benutzt, um einen (fast) vollständigen binären Baum mit Knotenmarkierung a zu einem Heap zu machen. In folgender Prozedur *buildheap* werden bottom-up alle Teilbäume mittels *downheap* zu Heaps gemacht. Statt bei den Blättern zu beginnen, die ja ohnehin bereits die Heap-Eigenschaft haben, genügt es, bei den inneren Knoten der Tiefe $d(T) - 1$ zu beginnen.

Prozedur *buildheap*

Eingabe: (Fast) vollständiger binärer Baum T mit Knotenmarkierung a

Ausgabe: Heap (durch Umordnung der Knotenmarken)

Methode: für $i = d(T) - 1$ abwärts bis 0

für alle inneren Knoten v der Tiefe $d(v) = i$

downheap(v);

Der folgende Algorithmus Heapsort konstruiert zunächst mittels *buildheap* aus den zu sortierenden Elementen einen Heap. Anschließend werden die Elemente in absteigender Reihenfolge ausgegeben, indem wie oben beschrieben jeweils die Markierung der Wurzel ausgegeben, durch die Markierung eines Blattes maximaler Tiefe ersetzt, das Blatt gelöscht und mit *downheap* die Heap-Struktur wiederhergestellt wird.

Algorithmus *Heapsort*

Eingabe: (Fast) vollständiger binärer Baum mit Wurzel r und Knotenmarkierung a

Ausgabe: Absteigend sortierte Folge der Knotenmarken

Methode: *buildheap*;

wiederhole solange r kein Blatt ist

gib $a(r)$ aus;

wähle Blatt b maximaler Tiefe;

markiere r mit $a(b)$;

lösche Blatt b;

downheap(r);

gib $a(r)$ aus;

Analyse

Ein (fast) vollständiger binärer Baum mit n Knoten hat die Tiefe $d \leq \log(n)$. Die Prozedur *downheap* benötigt daher höchstens $\log(n)$ Schritte.

Eine grobe Analyse ergibt, dass *buildheap* für höchstens n Knoten *downheap* aufruft. Heapsort ruft zusätzlich noch einmal für alle n Knoten *downheap* auf. Somit liegt die Zeitkomplexität von Heapsort in $O(n \cdot \log(n))$.

Eine genauere Analyse zeigt, dass *buildheap* sogar nur $O(n)$ Schritte benötigt. Denn im Verlauf der bottom-up-Konstruktion des Heaps wird *downheap* für höchstens $n/2$

Bäume der Tiefe 1, für höchstens $n/4$ Bäume der Tiefe 2 usw. und schließlich für einen Baum der Tiefe $\log(n)$ aufgerufen.

Somit benötigt *buildheap* höchstens

$$S = n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 3 + \ldots + 2 \cdot (\log(n) - 1) + 1 \cdot \log(n)$$

Schritte. Da

$$2S = n \cdot 1 + n/2 \cdot 2 + n/4 \cdot 3 + n/8 \cdot 4 + \ldots + 2 \cdot \log(n),$$

ergibt sich durch Subtraktion $2S - S$:

$$S = n + n/2 + n/4 + n/8 + \ldots + 2 - \log(n) .$$

Somit ist

$$S \leq 2n \in O(n).$$

Insgesamt beträgt die Zeitkomplexität von Heapsort allerdings trotzdem noch $T(n) \in O(n \log(n))$. Asymptotisch geht es nicht schneller, denn die untere Schranke für das Sortieren liegt bei $\Omega(n \log(n))$. Heapsort ist damit optimal, da es die untere Schranke erreicht (s. Abschnitt 2.8).

Implementierung

In einem Array a lässt sich ein (fast) vollständiger binärer Baum mit n Knoten und Knotenmarkierung sehr elegant speichern (Bild 2.5):

- die Wurzel steht an Position 0;
- die beiden Nachfolger des Knotens an Position v stehen an den Positionen $2v + 1$ und $2v + 2$.

Alle Knoten an den Positionen 0, ..., n div $2 - 1$ sind innere Knoten, alle Knoten an den Positionen n div 2, ..., $n - 1$ sind Blätter (div bezeichnet die ganzzahlige Division).

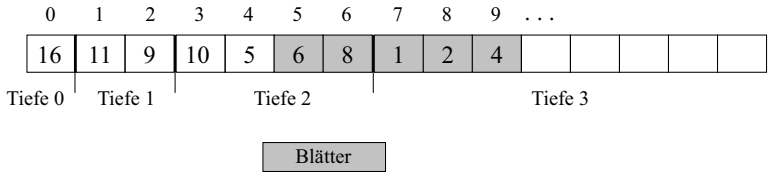

Bild 2.5: *Array-Repräsentation des Heaps von Bild 2.3*

Programm

Die folgende Klasse *HeapSorter* kapselt die Funktionen *downheap*, *buildheap* und *heapsort*. Die Methode *sort* übergibt das zu sortierende Array an das Array a und ruft *heapsort* auf.

In der hier angegebenen Implementierung der Funktion *heapsort* wird die jeweilige Knotenmarke der Wurzel nicht ausgegeben, sondern mit der Knotenmarke des zu löschenden Blattes vertauscht. Das Blatt wird „gelöscht“, indem die Anzahl der noch zu berücksichtigenden Knoten n um 1 vermindert wird. Auf diese Weise steht zum Schluss die aufsteigend sortierte Folge im Array.

Mit der Anweisung

```
HeapSorter s=new HeapSorter();
```

wird ein Objekt vom Typ *HeapSorter* erzeugt; danach kann mit

```
s.sort(b);
```

ein Array b sortiert werden. Es folgt das Programm.

```
public class HeapSorter
{
    private int[] a;
    private int n;

    public void sort(int[] a)
    {
        this.a=a;
        n=a.length;
        heapsort();
    }

    private void heapsort()
    {
        buildheap();
        while (n>1)
        {
            n--;
            exchange (0, n);
            downheap (0);
        }
    }

    private void buildheap()
    {
        for (int v=n/2-1; v>=0; v--)
            downheap (v);
    }
```

```
    private void downheap(int v)
    {
        int w=2*v+1;         // erster Nachfolger von v
        while (w<n)
        {
            if (w+1<n)       // gibt es einen zweiten Nachfolger?
                if (a[w+1]>a[w]) w++;
            // w ist der Nachfolger von v mit maximaler Markierung

            if (a[v]>=a[w]) return;  // v hat die Heap-Eigenschaft
            // sonst
            exchange(v, w);  // vertausche Markierungen von v und w
            v=w;             // fahre mit v=w fort
            w=2*v+1;
        }
    }

    private void exchange(int i, int j)
    {
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }

}    // end class HeapSorter
```

Variante: Bottomup-Heapsort

Mit der Komplexität von $\Theta(n \log(n))$ ist Heapsort optimal. Dennoch ist Quicksort im allgemeinen schneller als Heapsort. Die Variante *Bottomup-Heapsort* [Weg 93] spart gegenüber Standard-Heapsort etwa die Hälfte der Vergleiche ein.

Standard-Heapsort trägt nach dem Entnehmen des größten Elementes an der Wurzel die Markierung des letzten Blattes als neuen Wert an die freigewordene Position ein. Dieser Wert wird dann bei der Reorganisation des Heaps mit *downheap* wieder auf eine der (höchstwahrscheinlich) untersten Schichten durchgereicht. In jeder Schicht werden zwei Vergleiche durchgeführt. Einer der Vergleiche dient jeweils dazu, zu ermitteln, ob der durchgereichte Wert schon die richtige Position erreicht hat. Dies wird jedoch im allgemeinen zunächst nicht der Fall sein, da der Wert von einem Blatt stammt und daher klein ist.

Demgegenüber reicht Bottomup-Heapsort als erstes die freigewordene Position ganz nach unten durch. Hierfür ist nur ein Vergleich pro Schicht erforderlich. Dann wird die Markierung des letzten Blattes in die freigewordene Position eingetragen. Mithilfe der Prozedur *upheap* muss der Wert dann wieder so weit aufrücken, bis die Heap-Eigenschaft

wiederhergestellt ist. Da jedoch die Markierung des letzten Blattes im allgemeinen ein sehr kleiner Wert ist, sind hierfür, wenn überhaupt, nur wenige Schritte erforderlich.

Die folgenden Bilder zeigen die unterschiedlichen Vorgehensweisen. Bild 2.6 zeigt, wie Standard-Heapsort den neuen Wert von der Wurzel bis zu seiner richtigen Position durchreicht.

Bild 2.7 zeigt, wie Bottomup-Heapsort zunächst die an der Wurzel freigewordene Position bis ganz nach unten durchreicht (a) und wie der neue Wert dann gegebenenfalls wieder ein wenig aufrücken muss (b).

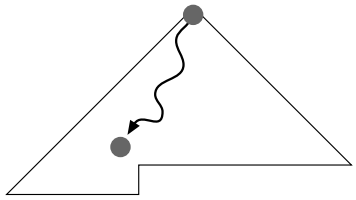

Bild 2.6: *Neuorganisieren des Heaps bei Standard-Heapsort*

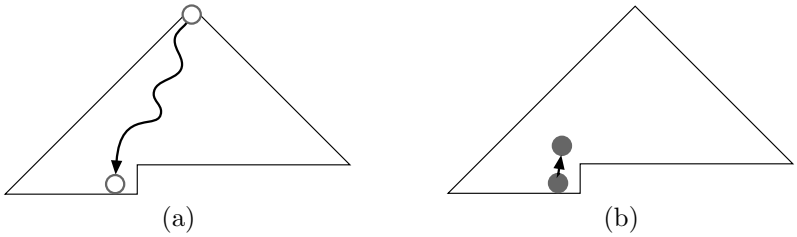

Bild 2.7: *Neuorganisieren des Heaps bei Bottomup-Heapsort*

Folgende Klasse *BottomupHeapSorter* stellt eine mögliche Implementierung der Idee von Bottomup-Heapsort dar. Die Funktionen *downheap* und *buildheap* sind dieselben wie in der Klasse *HeapSorter*.

```
public class BottomupHeapSorter
{
    private int[] a;
    private int n;
```

```
public void sort(int[] a)
{
    this.a=a;
    n=a.length;
    bottomupheapsort();
}

private void bottomupheapsort()
{
    int x, u;
    buildheap();
    while (n>1)
    {
        n--;
        x=a[n];    // Markierung des letzten Blatts
        a[n]=a[0];
        u=holedownheap();
        upheap(u, x);
    }
}

private void buildheap()
{
    for (int v=n/2-1; v>=0; v--)
        downheap (v);
}

private void downheap(int v)
{
    int w=2*v+1;         // erster Nachfolger von v
    while (w<n)
    {
        if (w+1<n)       // gibt es einen zweiten Nachfolger?
            if (a[w+1]>a[w]) w++;
        // w ist der Nachfolger von v mit maximaler Markierung
        if (a[v]>=a[w]) return;  // v hat die Heap-Eigenschaft
        // sonst
        exchange(v, w);  // vertausche Markierungen von v und w
        v=w;             // fahre mit v=w fort
        w=2*v+1;
    }
}

private int holedownheap()
{
    int v=0, w=2*v+1;
```

```
        while (w+1<n)     // zweiter Nachfolger existiert
        {
            if (a[w+1]>a[w]) w++;
            // w ist der Nachfolger von v mit maximaler Markierung
            a[v]=a[w];
            v=w; w=2*v+1;
        }
        if (w<n)    // einzelnes Blatt
        {
            a[v]=a[w];
            v=w;
        }
        // freigewordene Position ist an einem Blatt angekommen
        return v;
    }

    private void upheap(int v, int x)
    {
        int u;
        a[v]=x;
        while (v>0)
        {
            u=(v-1)/2;    // Vorgänger
            if (a[u]>=a[v]) return;
            // sonst
            exchange(u, v);
            v=u;
        }
    }

    private void exchange(int i, int j)
    {
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }

}    // end class BottomupHeapSorter
```

Zusammenfassung

Mit der Zeitkomplexität von $O(n \log(n))$ im schlechtesten Fall ist Heapsort optimal. Im Gegensatz zu Mergesort benötigt Heapsort keinen zusätzlichen Speicherplatz.

Gegenüber Quicksort ist Heapsort im Durchschnitt langsamer. Die Variante Bottomup-Heapsort kommt jedoch der Laufzeit von Quicksort sehr nahe.

2.4 Mergesort

Das Sortierverfahren *Mergesort* erzeugt eine sortierte Folge durch Verschmelzen (engl.: *to merge*) sortierter Teilstücke. Mit einer Zeitkomplexität von $\Theta(n \log(n))$ ist das Verfahren optimal.

Idee

Ähnlich wie Quicksort beruht das Verfahren auf dem Divide-and-Conquer-Prinzip (*teile und herrsche*). Die zu sortierende Folge wird zunächst in zwei Hälften aufgeteilt (*divide*), die jeweils für sich sortiert werden (*conquer*). Dann werden die sortierten Hälften zu einer insgesamt sortierten Folge verschmolzen (*combine*) (Bild 2.8).

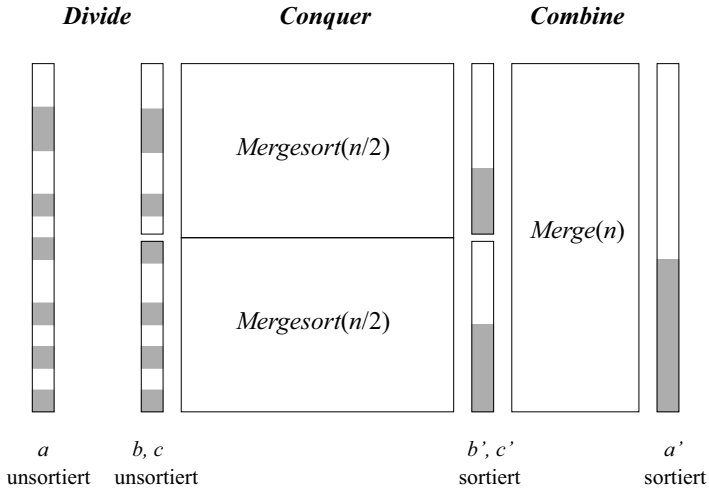

Bild 2.8: *Mergesort*(n)

Die folgende Funktion *mergesort* sortiert eine Folge a vom unteren Index lo bis zum oberen Index hi.

```
void mergesort(int lo, int hi)
{
    if (lo<hi)
    {
        int m=(lo+hi)/2;
        mergesort(lo, m);
        mergesort(m+1, hi);
        merge(lo, m, hi);
    }
}
```

Zunächst wird geprüft, ob die Folge aus mehr als einem Element besteht. Dies ist der Fall, wenn *lo* kleiner als *hi* ist. In diesem Fall wird als erstes die Mitte *m* zwischen *lo* und *hi* bestimmt. Anschließend wird die untere Hälfte der Folge (von *lo* bis *m*) sortiert und dann die obere Hälfte (von *m*+1 bis *hi*), und zwar durch rekursiven Aufruf von *mergesort*. Danach werden die sortierten Hälften durch Aufruf der Prozedur *merge* verschmolzen.

Die Funktionen *mergesort* und *merge* sind in eine Klasse *MergeSorter* eingebettet, in der das Datenarray *a* sowie ein benötigtes Hilfsarray *b* deklariert sind.

Die Hauptarbeit des Mergesort-Algorithmus liegt im Verschmelzen der sortierten Teilstücke, also in der Funktion *merge*. Es gibt unterschiedliche Ansätze für die Implementierung dieser Funktion.

a) Einfache Variante der Funktion *merge*

Diese Implementation der Funktion *merge* kopiert zunächst die beiden sortierten Hälften der Folge *a* hintereinander in eine neues Array *b*. Dann vergleicht sie die beiden Hälften mit einem Index *i* und einem Index *j* elementweise und kopiert jeweils das nächstgrößte Element zurück nach *a* (Bild 2.9).

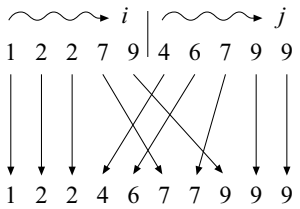

Bild 2.9: *Verschmelzen zweier sortierter Hälften*

Es kommt dabei am Ende zu der Situation, dass der eine Index am Ende seiner Hälfte angekommen ist, der andere Index aber noch nicht. Dann muss im Prinzip noch der Rest

der entsprechenden Hälfte zurückkopiert werden. Tatsächlich ist dies aber nur bei der vorderen Hälfte erforderlich. Bei der hinteren Hälfte stehen die betreffenden Elemente schon an Ort und Stelle.

Folgende Funktion *merge* ist eine mögliche Implementierung dieses Ansatzes.

```
// einfache Variante
void merge(int lo, int m, int hi)
{
    int i, j, k;

    // beide Hälften von a in Hilfsarray b kopieren
    for (i=lo; i<=hi; i++)
        b[i]=a[i];

    i=lo; j=m+1; k=lo;
    // jeweils das nächstgrößte Element zurückkopieren
    while (i<=m && j<=hi)
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j++];

    // Rest der vorderen Hälfte falls vorhanden zurückkopieren
    while (i<=m)
        a[k++]=b[i++];
}
```

In Java ist die Kurzschreibweise `k++` gleichbedeutend mit `k=k+1`; die Anweisung `a[k++]=b[i++]` ist äquivalent zu der Anweisungsfolge `a[k]=b[i]; k=k+1; i=i+1`.

b) Effizientere Variante der Funktion *merge*

Tatsächlich genügt es, nur die vordere Hälfte der Folge in ein neues Array b auszulagern. Die hintere Hälfte bleibt an Ort und Stelle im Array a. Dadurch wird nur halb soviel zusätzlicher Speicherplatz und nur halb soviel Zeit zum Kopieren benötigt [Som 04].

Beim Zurückkopieren wird indirekt überwacht, ob die Elemente des Arrays b schon fertig zurückkopiert sind; dies ist der Fall, wenn der Index k den Index j erreicht. Ähnlich wie in Variante a befindet sich ein dann möglicherweise noch verbleibender Rest der hinteren Hälfte bereits an Ort und Stelle.

Dieser Ansatz ist in folgender Implementation der Funktion *merge* verwirklicht.

```
// effizientere Variante
void merge(int lo, int m, int hi)
{
    int i, j, k;
```

```
    i=0; j=lo;
    // vordere Hälfte von a in Hilfsarray b kopieren
    while (j<=m)
        b[i++]=a[j++];

    i=0; k=lo;
    // jeweils das nächstgrößte Element zurückkopieren
    while (k<j && j<=hi)
        if (b[i]<=a[j])
            a[k++]=b[i++];
        else
            a[k++]=a[j++];

    // Rest von b falls vorhanden zurückkopieren
    while (k<j)
        a[k++]=b[i++];
}
```

c) Bitonische Variante der Funktion *merge*

Bei diese Variante der Funktion *merge* wird die vordere Hälfte der Folge in ihrer normalen Reihenfolge, die hintere Hälfte jedoch in umgekehrter Reihenfolge in die Folge b kopiert [Sed 88]. Dadurch entsteht eine zunächst ansteigende und dann abfallende Folge (eine sogenannte bitonische Folge).

Die Prozedur *merge* durchläuft nun mit dem Index i die vordere Hälfte von links nach rechts und mit dem Index j die hintere Hälfte von rechts nach links. Das jeweils nächstgrößte Folgenelement wird in die Folge a zurückkopiert. Die gesamte Folge ist fertig zurückkopiert, wenn i und j sich überkreuzen, d.h. wenn $i > j$ wird (Bild 2.10). Es ist dabei nicht notwendig, dass i und j in „ihren“ Hälften bleiben.

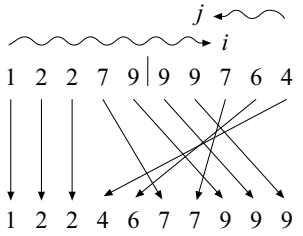

Bild 2.10: *Verschmelzen zweier gegenläufig sortierter Hälften*

Die folgende Version der Funktion *merge* realisiert diesen Ansatz.