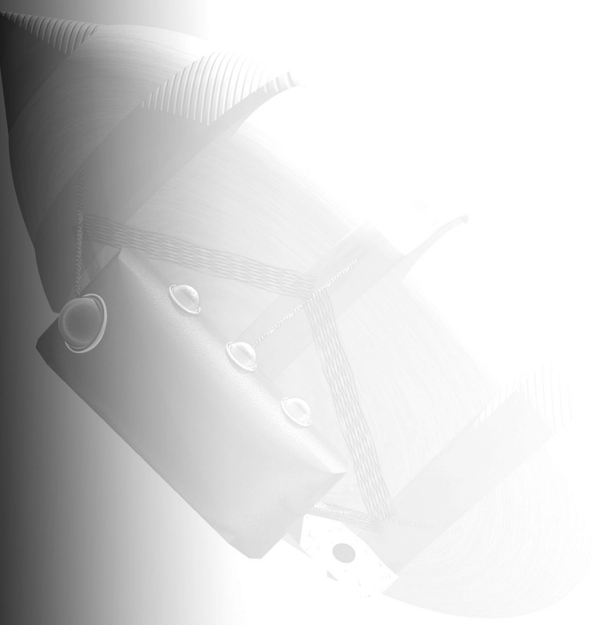
Teil 3
Sortieren



Kapitel

6 Elementare Sortierverfahren

Bei unserem ersten Ausflug in das Gebiet der Sortieralgorithmen untersuchen wir mehrere elementare Verfahren, die für kleine Dateien oder für Dateien, die eine spezielle Struktur haben, geeignet sind. Es gibt mehrere Gründe, um diese einfachen Sortieralgorithmen umfassend zu untersuchen. Erstens bieten sie einen Kontext, in dem wir uns mit der Terminologie und den grundlegenden Mechanismen für Sortieralgorithmen vertraut machen können und uns somit erlauben, einen angemessenen Hintergrund für das Studium komplizierterer Algorithmen zu entwickeln. Zweitens sind diese einfachen Verfahren in vielen Sortieranwendungen tatsächlich effizienter als die leistungsfähigeren Universalverfahren. Und drittens lassen sich mehrere der einfachen Verfahren zu besseren Universalverfahren erweitern oder sind nützlich, wenn man die Effizienz anspruchsvollerer Verfahren verbessern will.

In diesem Kapitel wollen wir nicht nur die elementaren Verfahren einführen, sondern auch einen Rahmen entwickeln, in dem wir das Sortieren in späteren Kapiteln untersuchen können. Wir sehen uns eine Vielfalt von Situationen an, die für die Anwendung von Sortieralgorithmen wichtig sein können, untersuchen unterschiedliche Arten von Eingabedateien und werfen einen Blick auf andere Möglichkeiten, um Sortierverfahren zu vergleichen und ihre Eigenschaften kennen zu lernen.

Zuerst sehen wir uns ein einfaches Testprogramm für Sortierverfahren an, das einen Kontext für die Konventionen bietet, an die wir uns halten sollten. Außerdem betrachten wir die grundlegenden Eigenschaften von Sortierverfahren, die wir kennen müssen, wenn wir die Eignung von Algorithmen für bestimmte Anwendungen beurteilen. Als Nächstes tauchen wir in ein Thema ein, das wir bereits in den Kapiteln 3 und 4 angeschnitten haben: die Entwicklung von Schnittstellen und Implementierungen für Datentypen. Damit erweitern wir unsere Algorithmen, um auch solche Datendateien sortieren zu können, wie man sie in der Praxis vorfindet. Dann beschäftigen wir uns eingehend mit Implementierungen der drei elementaren Verfahren: Sortieren durch Auswählen, Sortieren durch Einfügen und Bubblesort. Daran anschließend untersuchen wir die Leistungsdaten dieser Algorithmen im Detail. Als Nächstes behandeln wir Shellsort, der vielleicht nicht zu den elementaren Algorithmen zu rechnen ist, aber leicht zu implementieren und eng mit dem Sortieren durch Einfügen verwandt ist. Nach einem Ausflug in die mathematischen Eigenschaften von Shellsort betrachten wir das Sortieren mit verketteten Listen. Zum Schluss behandelt dieses Kapitel ein spezielles Verfahren, das sich anbietet, wenn Schlüsselwerte von vornherein auf einen kleinen Bereich beschränkt sind.

In zahlreichen Sortieranwendungen kann ein einfacher Algorithmus das bevorzugte Verfahren sein. Erstens setzen wir oftmals ein Sortierprogramm nur einmal oder wenige Male ein. Nachdem wir ein Sortierproblem für eine Datenmenge »gelöst« haben, müssen wir nicht wieder auf das Sortierprogramm in der Anwendung zurückgreifen, die diese Daten manipuliert. Wenn ein elementarer Sortiervorgang nicht langsamer als irgendein

anderer Teil der Datenverarbeitung ist – beispielsweise das Einlesen der Daten oder deren Ausgabe –, dann gibt es keinen Grund, um nach einer schnelleren Variante zu suchen. Ist die Anzahl der zu sortierenden Elemente nicht zu groß (sagen wir weniger als einige Hundert Elemente), können wir uns ohne weiteres dafür entscheiden, ein einfaches Verfahren zu implementieren und auszuführen, anstatt uns mit der Schnittstelle zu einem Systemsortierprogramm oder mit dem Implementieren und Debuggen eines komplizierten Verfahrens aufzuhalten. Zweitens sind elementare Verfahren immer geeignet für kleine Dateien (sagen wir weniger als einige Dutzend Elemente) – intelligentere Algorithmen bringen im Allgemeinen Overhead mit sich, der sie bei kleinen Dateien langsamer als die elementaren Algorithmen macht. Diesen Fragen brauchen wir uns nicht zuzuwenden, sofern wir nicht viele kleine Dateien sortieren wollen. Allerdings sind Anwendungen mit derartigen Anforderungen nichts Ungewöhnliches. Auch Dateien, die fast sortiert (oder überhaupt schon sortiert!) vorliegen oder die eine große Anzahl von doppelten Schlüsseln enthalten, lassen sich leicht sortieren. Es zeigt sich, dass einige einfache Verfahren besonders effizient laufen, wenn solche gut strukturierten Dateien zu sortieren sind.

Es gilt die Regel, dass die hier behandelten elementaren Verfahren eine Laufzeit proportional zu N^2 benötigen, um N zufällig angeordnete Elemente zu sortieren. Für kleine N ist die Laufzeit meistens noch akzeptabel. Wie bereits erwähnt, sind die Verfahren für kleine Dateien und in anderen speziellen Situationen höchstwahrscheinlich sogar schneller als hoch entwickelte Verfahren. Jedoch sind die in diesem Kapitel behandelten Verfahren *nicht* für große, zufällig angeordnete Dateien geeignet, weil die Laufzeit selbst auf den schnellsten Computern übermäßig zunimmt. Eine bemerkenswerte Ausnahme ist Shellsort (siehe Abschnitt 6.6), der weniger als N^2 Schritte für große N benötigt und vermutlich das bevorzugte Verfahren für mittelgroße Dateien und für einige andere Spezialanwendungen darstellt.

6.1 Spielregeln

Bevor wir uns spezifischen Algorithmen zuwenden, wollen wir zunächst allgemeine Fachbegriffe und grundlegende Annahmen für Sortieralgorithmen klären. Wir betrachten Verfahren zum Sortieren von *Dateien* (Files) mit *Elementen* (Items), die *Schlüssel* (Keys) enthalten. Alle diese Konzepte sind natürliche Abstraktionen in modernen Programmierumgebungen. Die Schlüssel, die einen Teil der Elemente darstellen (oftmals nur einen kleinen Teil), dienen dazu, das Sortieren zu steuern. Das Sortierverfahren soll die Elemente neu anordnen, so dass ihre Schlüssel entsprechend einer klar definierten Sortiervorschrift (gewöhnlich in numerischer oder alphabetischer Reihenfolge) sortiert sind. Spezielle Eigenschaften der Schlüssel und der Elemente können über die Anwendungen hinweg stark variieren, wobei jedoch das Sortierproblem immer durch das abstrakte Konzept, Schlüssel und zugeordnete Informationen in eine Reihenfolge zu bringen, charakterisiert ist.

Wenn sich die zu sortierende Datei im Hauptspeicher unterbringen lässt, sprechen wir von einem *internen* Sortierverfahren. Das Sortieren von Dateien, die auf einem Magnetband

oder auf einer Festplatte gespeichert sind, heißt *externes* Sortieren. Der wesentliche Unterschied zwischen beiden besteht darin, dass ein internes Sortierverfahren leicht auf jedes beliebige Element zugreifen kann, während ein externes Sortierverfahren nur sequenziell oder höchstens in Form von Blöcken auf die Elemente zugreifen kann. Auch wenn wir uns in Kapitel 11 einige externe Sortierverfahren ansehen, beschäftigen wir uns hier hauptsächlich mit internen Sortierverfahren.

Wir sehen uns auch Arrays und verkettete Listen an. Sowohl das Sortieren von Arrays als auch das Sortieren von verketteten Listen ist von Interesse: Während wir unsere Algorithmen entwickeln, stellen sich grundlegende Aufgaben, die am besten für einen sequenziellen Speicherbereich geeignet sind, und andere Aufgaben, die sich für verkettete Speicherbereiche anbieten. Einige dieser klassischen Verfahren sind ausreichend abstrakt, sodass sie sich sowohl für Arrays als auch für verkettete Listen effizient implementieren lassen; andere sind entweder für die eine oder die andere Variante prädestiniert. Von Interesse sind auch andere Arten der Zugriffseinschränkungen.

Als Erstes konzentrieren wir uns auf das Sortieren von Arrays. Programm 6.1 veranschaulicht viele der Konventionen, die wir in unseren Implementierungen verwenden. Es besteht aus einem Testprogramm, das ein Array mit Ganzzahlen füllt, die (abhängig von einem ganzzahligen Argument) entweder von der Standardeingabe gelesen oder zufällig generiert werden; dann ruft es eine Sortiermethode auf, um die Ganzzahlen im Array zu ordnen; schließlich gibt es das sortierte Ergebnis aus.

Programm 6.1
Beispiel für das Sortieren eines Arrays mit Testprogramm

Dieses Programm veranschaulicht unsere Konventionen für die Implementierung grundsätzlicher Sortierverfahren für Arrays. Die Methode `main` stellt eine Testroutine dar, die ein Array von Gleitkommazahlen (`double`) mit zufälligen Werten initialisiert, eine Methode `sort` aufruft, die dieses Array sortiert, und dann die geordneten Ergebnisse ausgibt.

Die hier angegebene Methode `sort` ist eine Version für das Sortieren durch Einfügen (Abschnitt 6.4 bringt eine umfassende Beschreibung, ein Beispiel und eine verbesserte Implementierung). Sie verwendet die Methoden `less` (die zwei Elemente vergleicht), `exch` (die zwei Elemente vertauscht) und `compExch` (die zwei Elemente vergleicht und sie gegebenenfalls vertauscht, damit das zweite Elemente nicht kleiner als das erste ist).

Mit diesem Code können wir Arrays jedes elementaren numerischen Typs sortieren, indem wir an allen Stellen `double` durch den jeweiligen Typnamen ersetzen. Mit einer geeigneten Implementierung von `less` lassen sich Arrays von Elementen eines beliebigen Referenztyps sortieren (siehe Abschnitt 6.2).

```
class ArraySortBasic
  { static int cnt = 0;
    static boolean less(double v, double w)
      { cnt++; return v < w; }
```

→

```
  static void exch(double[] a, int i, int j)
    { double t = a[i]; a[i] = a[j]; a[j] = t; }
  static void compExch(double[] a, int i, int j)
    { if (less(a[j], a[i])) exch (a, i, j); }
  static void sort(double[] a, int l, int r)
    { for (int i = l+1; i <= r; i++)
        for (int j = i; j > l; j--)
          compExch(a, j-1, j);
    }
  public static void main(String[] args)
    { int N = Integer.parseInt(args[0]);
      double a[] = new double[N];
      for (int i = 0; i < N; i++)
        a[i] = Math.random();
      sort(a, 0, N-1);
      if (N < 100)
        for (int i = 0; i < N; i++)
          Out.println(a[i] + "");
      Out.println("Vergleiche: " + cnt);
    }
}
```

Wie wir aus den Kapiteln 3 und 4 wissen, gibt es zahlreiche Mechanismen, mit denen wir unsere Sortierimplementierungen für andere Datentypen anpassen können. Im Detail geht Abschnitt 6.2 auf solche Mechanismen ein. Die Methode `sort` in Programm 6.1 verweist auf die zu sortierenden Elemente nur über ihren ersten Parameter und wenige einfache Datenoperationen. Wie üblich erlaubt uns dieser Ansatz, denselben Code für das Sortieren anderer Typen von Elementen zu nutzen. Wenn wir beispielsweise den Code für das Generieren, Speichern und Ausgeben zufälliger Schlüssel in der Methode `main` von Programm 6.1 ändern, um Ganzzahlen anstelle von Gleitkommazahlen zu verarbeiten, müssen wir lediglich in der Methode `sort` und ihren zugehörigen Methoden `double` in `int` ändern, um Arrays von Ganzzahlen sortieren zu lassen. Um diese Flexibilität zu erhalten (während wir gleichzeitig explizit die Variablen kennzeichnen, die Elemente aufnehmen) werden unsere Sortierimplementierungen mithilfe eines generischen Datentyps `ITEM` implementiert. Fürs Erste können wir uns `ITEM` als `int` oder `double` vorstellen; in Abschnitt 6.2 behandeln wir Datentypimplementierungen, mit denen wir unsere Sortierprogramme für beliebige Elemente verwenden können – unter anderem für Gleitkommazahlen, Strings und andere Arten von Schlüsseln. Dazu greifen wir auf die Mechanismen zurück, die die Kapitel 3 und 4 vorgestellt haben.

Die Methode `sort` können wir durch jede der Implementierungen für das Sortieren von Arrays aus diesem Kapitel und aus den Kapiteln 7 bis 10 ersetzen. Alle gehen davon aus, dass Elemente vom Typ `ITEM` zu sortieren sind, und alle übernehmen drei Parameter: das Array sowie die linke und rechte Grenze des zu sortierenden Teilarrays. Außerdem stützen sich alle Implementierungen auf die Methode `less`, um Schlüssel in Elementen zu vergleichen, und `exch` oder `compExch`, um Elemente zu vertauschen. Um die Sortierverfahren auseinander zu halten, geben wir den verschiedenen Sortierroutinen unterschiedliche Namen. Man

muss lediglich eine von ihnen umbenennen oder das Testprogramm anpassen, um zwischen den Algorithmen in einem Clientprogramm wie zum Beispiel Programm 6.1 zu wechseln, ohne irgendwelchen Code in der Sortierimplementierung ändern zu müssen.

Diese Konventionen gestatten uns, Implementierungen für viele Algorithmen zum Sortieren von Arrays natürlich und knapp zu untersuchen. Das Testprogramm in Abschnitten 6.2 veranschaulicht, wie man die Implementierungen in allgemeineren Kontexten einsetzen kann. Außerdem zeigt dieser Abschnitt mehrere Implementierungen von Datentypen. Obwohl wir immer auf solche Rahmenbetrachtungen achten, konzentrieren wir uns in erster Linie auf algorithmische Fragen, auf die wir nun zu sprechen kommen.

Die Beispielmethode `sort` in Programm 6.1 ist eine Variante des Verfahrens *Sortieren durch Einfügen*, auf das wir in Abschnitt 6.4 umfassend eingehen. Da es nur mit den Operationen *Vergleichen* und *Vertauschen* arbeitet, ist es ein Beispiel für ein *nichtadaptives* Sortieren: Die Sequenz der von ihm ausgeführten Operationen ist unabhängig von der Reihenfolge der Daten. Im Gegensatz dazu führt ein *adaptives* Sortieren unterschiedliche Sequenzen von Operationen aus, die von den Vergleichsergebnissen (Aufrufen von `less`) abhängig sind. Zwar sind nichtadaptive Sortierverfahren interessant, weil sie sich für Hardwareimplementierungen eignen (siehe Kapitel 11), die meisten hier betrachteten Universalsortierverfahren sind jedoch adaptiv.

Wie üblich steht von den Leistungsparametern vor allem die Laufzeit unserer Sortieralgorithmen im Mittelpunkt. Die Methode `sort` in Programm 6.1 führt immer genau $N(N-1)/2$ Vergleiche (und Vertauschungen) aus, sodass sie sich nicht verwenden lässt, wenn N sehr groß ist. Die Verfahren *Sortieren durch Auswählen*, *Sortieren durch Einfügen* und *Bubblesort*, die wir in den Abschnitten 6.3 bis 6.5 behandeln, erfordern eine Laufzeit proportional zu N^2, um N Elemente zu sortieren, wie es Abschnitt 6.6 erläutert. Die erweiterten Verfahren, die wir in den Kapiteln 7 bis 10 besprechen, können N Elemente in einer Laufzeit proportional zu $N \log N$ sortieren. Allerdings sind diese Verfahren für kleine N und in bestimmten anderen Spezialfällen nicht immer so gut wie die hier betrachteten Verfahren. In Abschnitt 6.8 sehen wir uns ein anspruchsvolleres Verfahren (Shellsort) an, das eine Zeit proportional zu $N^{3/2}$ oder weniger benötigt, und in Abschnitt 6.10 geht es um ein spezielles Verfahren (schlüsselindiziertes Sortieren), das für bestimmte Arten von Schlüsseln eine Laufzeit proportional zu N aufweist.

Die in den vorherigen Absätzen beschriebenen analytischen Ergebnisse folgen alle aus der numerischen Auswertung der grundlegenden Operationen (Vergleichen und Vertauschen), die diese Algorithmen durchführen. Wie Abschnitt 2.2 erläutert hat, müssen wir auch die Kosten dieser Operationen berücksichtigen, und es lohnt sich im Allgemeinen, sich auf die am häufigsten ausgeführten Operationen (die innere Schleife des Algorithmus) zu konzentrieren. Unser Ziel ist es, effiziente und vernünftige Implementierungen effizienter Algorithmen zu entwickeln. Um dieses Ziel zu erreichen, verzichten wir nicht einfach auf überflüssige Elemente in den inneren Schleifen, sondern suchen auch nach Wegen, um Anweisungen aus inneren Schleifen nach Möglichkeit zu entfernen. Im Allgemeinen lassen sich die Kosten in einer Anwendung am besten verringern, wenn man zu einem effizienteren Algorithmus wechselt; die zweitbeste Methode ist, die innere Schleife zu straffen. Beide Optionen sehen wir uns eingehend bei Sortieralgorithmen an.

Adams	1
Black	2
Brown	4
Jackson	2
Jones	4
Smith	1
Thompson	4
Washington	2
White	3
Wilson	3

Adams	1
Smith	1
Washington	2
Jackson	2
Black	2
White	3
Wilson	3
Thompson	4
Brown	4
Jones	4

Adams	1
Smith	1
Black	2
Jackson	2
Washington	2
White	3
Wilson	3
Brown	4
Jones	4
Thompson	4

Abbildung 6.1:
Beispiel für ein stabiles Sortieren

Ein Sortieren dieser Datensätze kann für jeden Schlüssel geeignet sein. Nehmen wir an, dass sie anfangs nach dem ersten Schlüssel sortiert werden (oben). Ein nichtstabiles Sortierverfahren auf dem zweiten Schlüssel bewahrt die Reihenfolge in Datensätzen mit doppelten Schlüsseln nicht (Mitte), während ein stabiles Sortierverfahren die Reihenfolge beibehält (unten).

Der Umfang an zusätzlichem Speicherplatz, den ein Sortieralgorithmus beansprucht, ist der zweiwichtigste Faktor, den wir berücksichtigen müssen. Grundsätzlich kann man die Verfahren in drei Typen einteilen: (1.) diejenigen, die an Ort und Stelle (in situ) sortieren und keinen zusätzlichen Speicherplatz außer vielleicht für einen kleinen Stack oder eine Tabelle benötigen, (2.) diejenigen, die eine Darstellung mit verketteter Liste verwenden oder anderweitig auf Daten über Referenzen oder Arrayindizes verweisen und so zusätzlichen Speicherplatz für N Referenzen oder Indizes benötigen, und (3.) diejenigen, die so viel zusätzlichen Speicher benötigen, um eine Kopie des zu sortierenden Arrays aufnehmen zu können.

Häufig verwenden wir Sortierverfahren für Elemente mit mehrfachen Schlüsseln – es kann sogar sein, dass wir eine Gruppe von Elementen mit unterschiedlichen Schlüsseln zu unterschiedlichen Zeitpunkten sortieren müssen. In derartigen Fällen kann es für uns wichtig sein zu wissen, ob das Sortierverfahren die folgende Eigenschaft aufweist:

Definition 6.1

Ein Sortierverfahren ist stabil, wenn es die relative Reihenfolge der Elemente mit doppelten Schlüsseln in der Datei bewahrt. ■

Wenn zum Beispiel eine alphabetische Liste von Studenten und ihrem Studienjahr nach dem Jahr sortiert ist, produziert ein stabiles Verfahren eine Liste, in der die Personen in derselben Jahrgangsstufe immer noch in alphabetischer Reihenfolge stehen, während ein nichtstabiles Verfahren möglicherweise eine Liste produziert, die keine Spur auf die ursprüngliche alphabetische Reihenfolge zurücklässt. Abbildung 6.1 zeigt dazu ein Beispiel. Oftmals sind Programmierer, die mit der Stabilität nicht vertraut sind, überrascht, wie ein unstabiler Algorithmus die Daten scheinbar durcheinander bringt, wenn sie erstmals mit dieser Situation zu tun haben.

Einige (aber nicht alle) der einfachen Sortierverfahren, die wir in diesem Kapitel betrachten, sind stabil.

Andererseits sind viele (aber nicht alle) der komplizierteren Algorithmen, denen wir in den nächsten Kapiteln begegnen, nicht stabil. Wenn die Stabilität im Vordergrund steht, können wir sie erzwingen, indem wir vor dem Sortieren einen kleinen Index an jeden Schlüssel anfügen oder die Schlüssellänge in irgendeiner anderen Form vergrößern. Dieser zusätzliche Aufwand ist gleichbedeutend mit der Verwendung beider Schlüssel für das Sortieren in Abbildung 6.1; hier ist ein stabiler Algorithmus vorzuziehen. Es ist leicht, die Stabilität als gegeben vorauszusetzen; tatsächlich aber erreichen nur wenige der hoch entwickelten Verfahren, die wir in späteren Kapiteln kennen lernen, diese Stabilität ohne zusätzlichen Zeit- oder Speicherplatzaufwand.

Wie bereits erwähnt, greifen Sortierprogramme normalerweise auf die Elemente nach einer der folgenden beiden Methoden zu: entweder werden die Schlüssel verglichen oder die gesamten Elemente werden verschoben. Wenn die zu sortierenden Elemente groß sind, empfiehlt es sich, das Umspeichern mithilfe eines indirekten Sortierens zu vermeiden: Wir ordnen nicht die Elemente selbst neu an, sondern nur ein Array von Referenzen, sodass der erste Eintrag auf das kleinste Element verweist, der zweite Eintrag auf das nächstkleinere Element usw. Nach dem Sortieren könnten wir die Elemente umordnen, was aber oftmals unnötig ist, weil wir auf sie (indirekt) in sortierter Reihenfolge zugreifen können. In Java ist das indirekte Sortieren tatsächlich die Norm, wie der nächste Abschnitt zeigt, in dem wir uns dem Sortieren von Dateien mit Elementen widmen, die keine einfachen numerischen Typen sind.

ÜBUNGEN

▷ 6.1 Das Sortierspielzeug eines Kindes hat *i* Ringe, die auf jeweils einen Stab an der Position *i* für *i* von 1 bis 5 passen. Geben Sie das Verfahren an, nachdem Sie die Ringe auf die Stäbe legen. Nehmen Sie an, dass Sie anhand des Ringes nicht im Voraus feststellen können, ob er auf einen Stab passt (Sie müssen erst probieren, ob er passt).

6.2 Ein Kartentrick verlangt, dass Sie einen Kartenstapel nach der Farbe auflegen (in der Reihenfolge Eichel, Grün, Rot, Schellen) und innerhalb jeder Farbe nach dem Wert. Bitten Sie ein paar Bekannte, diese Aufgabe auszuführen (Mischen nicht vergessen!) und geben Sie die Methode(n) an, die sie verwenden.

6.3 Erklären Sie, wie Sie einen Kartenstapel sortieren, wobei die Einschränkung gilt, dass die Karten mit dem Bild nach unten in einer Reihe aufzulegen sind. Es ist nur erlaubt, die Werte zweier Karten anzusehen und die Karten (optional) zu vertauschen.

○ 6.4 Erläutern Sie, wie Sie einen Kartenstapel sortieren, wobei die Einschränkung gilt, dass die Karten auf dem Stapel bleiben müssen. Es sind nur folgende Operationen erlaubt: die Werte der obersten beiden Karten auf dem Stapel ansehen, die obersten beiden Karten vertauschen und die oberste Karte ganz nach unten legen.

6.5 Geben Sie alle Sequenzen von drei Vergleichen-Vertauschen-Operationen an, die drei Elemente sortieren (siehe Programm 6.1).

○ 6.6 Geben Sie eine Sequenz von fünf Vergleichen-Vertauschen-Operationen an, die vier Elemente sortieren.

→

➔ 6.7 Die Überprüfung, ob das Array nach Aufruf von `sort` sortiert ist, liefert keine Garantie, dass das Sortieren funktioniert. Warum nicht?

- 6.8 Schreiben Sie ein Clientprogramm, das die Leistung testet und dazu mehrmals `sort` auf Dateien verschiedener Größen aufruft, die Zeit für jeden Lauf misst und die durchschnittliche Laufzeit anzeigt oder als Diagramm zeichnet.
- 6.9 Schreiben Sie ein Clientprogramm, das `sort` für schwierige oder extreme Fälle aufruft, die aber in praktischen Anwendungen durchaus auftreten können. Beispiele dafür sind sortierte Dateien, umgekehrt sortierte Dateien, Dateien mit durchweg gleichen Schlüsseln, Dateien mit nur zwei unterscheidbaren Werten und Dateien der Größe 0 oder 1.

6.2 Generische Sortierimplementierungen

Auch wenn es bei den meisten Sortieralgorithmen sinnvoll ist, sie sich beim Studium einfach so vorzustellen, dass sie Arrays von Zahlen in eine numerische Reihenfolge oder Zeichen in eine alphabetische Reihenfolge bringen, ist es ebenso angebracht zu erkennen, dass die Algorithmen größtenteils unabhängig vom Typ der sortierten Elemente sind und dass es nicht schwierig ist, zu einer allgemeinelren Darstellung überzugehen. In diesem Abschnitt behandeln wir die Konventionen, die wir beachten sollten, um unsere Sortierimplementierungen in einer Vielfalt von Kontexten nutzbringend einsetzen zu können. Indem wir uns gleich zu Anfang diesem Thema widmen, können wir die Einsatzbreite unseres Codes beträchtlich erweitern und seine Handhabung erleichtern.

Programm 6.2 definiert die wichtigsten Konventionen: um Elemente zu sortieren, müssen wir sie vergleichen können. In Java drücken wir eine derartige Anforderung aus, indem wir eine Schnittstelle definieren, die die benötigte Methode einbindet, und verlangen, dass jede Klasse, die ein zu sortierendes Element definiert, diese Schnittstelle implementiert. Derartige Implementierungen sind nicht schwer zu entwickeln (später in diesem Abschnitt betrachten wir drei detaillierte Beispiele in den Programmen 6.8 bis 6.11); im Ergebnis können wir die Methode `less` für Vergleiche in unseren Implementierungen verwenden und dennoch mit ihnen alle Typen von Elementen sortieren. Diese Schnittstelle ähnelt weitgehend der Java-Schnittstelle `Comparable`, die von den Implementierungen verlangt, dass sie die Methode `compareTo` einbinden.

Programm 6.2 Elementschnittstelle

Jede Klasse von Elementen, die zu sortieren sind, muss eine Methode einbinden, über die ein Objekt der Klasse bestimmen kann, ob es kleiner als ein anderes Objekt der Klasse ist oder nicht.

```
interface ITEM
  { boolean less(ITEM v); }
```

Über diesen fundamentalen Schritt hinaus lassen wir algorithmische Details auf dem Weg zur Entwicklung generischer Implementierungen zunächst außer Acht; Sie dürfen getrost den Rest dieses Abschnitts überspringen und sich später damit beschäftigen, um zunächst die grundlegenden Algorithmen und deren Eigenschaften in den Abschnitten 6.3 bis 6.6 und 6.8 sowie in den Kapiteln 7 bis 9 kennen zu lernen. Die betreffenden Implementierungen können Sie als Ersatz für die Methode `sort` in Programm 6.1 auffassen, die den elementaren Typ `double` anstelle des generischen Typs `ITEM` verwendet.

Programm 6.3 zeigt unsere Konvention zur Verpackung von Sortierimplementierungen als Methode in der Klasse `Sort`. Jeder Client mit einem Array von Objekten einer Klasse, die die Schnittstelle `ITEM` implementiert (und demzufolge eine Methode `less` definiert), kann das Array durch Aufruf von `Sort.sort` sortieren. In diesem Buch sehen wir uns zahlreiche Implementierungen von `sort` an. Um Verwechslungen zu vermeiden, geben wir jeder Implementierung einen anderen Namen, sodass Clients einfach `sort` aufrufen, wir aber eine andere Implementierung substituieren können, indem wir entweder den Code für die Methode `example` in dieser `Sort.java`-Datei austauschen oder auf den Java-Mechanismus des Klassenpfads zurückgreifen (siehe Abschnitt 4.6).

Programm 6.3 Klasse für Sortiermethoden

Unseren eigentlichen `sort`-Code bringen wir als statische Methode in einer separaten Klasse unter, sodass jeder Client sie verwenden kann, um ein beliebiges Array von Objekten eines beliebigen Typs zu sortieren, der die Methode `less` gemäß der `ITEM`-Schnittstellenspezifikation implementiert.

Diese Klasse bietet sich auch an, um die statischen Hilfsmethoden `less`, `exch` und `compExch` für unsere Sortierimplementierungen bereitzustellen.

```
class Sort
  {
    static boolean less(ITEM v, ITEM w)
      { return v.less(w); }
    static void exch(ITEM[] a, int i, int j)
      { ITEM t = a[i]; a[i] = a[j]; a[j] = t; }
    static void compExch(ITEM[] a, int i, int j)
      { if (less(a[j], a[i])) exch (a, i, j); }
    static void sort(ITEM[] a, int l, int r)
      { example(a, l, r); }
    static void example(ITEM[] a, int l, int r)
      {
        for (int i = l+1; i <= r; i++)
          for (int j = i; j > l; j--)
            compExch(a, j-1, j);
      }
  }
```

Unsere Sortierimplementierungen verwenden im Allgemeinen eine statische Methode `less` mit zwei Parametern, um Elemente zu vergleichen. Für Elemente einfacher Typen können wir den Operator < wie in Programm 6.1 verwenden. Sind die Elemente von einer Klasse, die `ITEM` implementiert, können wir `less` als Klassenmethode `less` für diese Klasse definieren, wie es Programm 6.3 zeigt. Einige unserer Sortierimplementierungen verwenden auch die Methoden `exch` und `compExch`, deren Implementierungen für einfache Elemente in Programm 6.1 und für Referenzelemente in Programm 6.3 angegeben sind. Alternativ könnten wir `less` in unsere Elementklasse und die Austauschoperationen in unsere Arrayklasse aufnehmen oder sie – da es sich jeweils nur um eine Codezeile handelt – einfach als private Methoden in die Sortierroutinen einbinden. Durch diese Methoden sind wir flexibel, um beispielsweise unsere Sortierroutinen zu erweitern und mit einer Methode die Anzahl der Vergleiche zu ermitteln (wie in Programm 6.1) oder die Algorithmen zu animieren (wie in Programm 6.16).

Wir haben schon eingehend darüber gesprochen, wie wir unsere Programme in unabhängige Module aufteilen, um Datentypen und abstrakte Datentypen zu implementieren (siehe die Kapitel 3 und 4); in diesem Abschnitt wenden wir die erwähnten Konzepte an, um Implementierungen, Schnittstellen und Clientprogramme für Sortieralgorithmen zu erstellen. Insbesondere betrachten wir abstrakte Datentypen für

- *Elemente* oder generische Objekte, die zu sortieren sind
- *Arrays* von Elementen

Durch den Elementdatentyp sind wir in der Lage, unseren Sortiercode für jeden Datentyp, der bestimmte Basisoperationen definiert, zu verwenden. Dieser Ansatz ist sowohl für einfache Typen wie auch für Referenztypen effizient und wir sehen uns zahlreiche Implementierungen an. Die Arrayschnittstelle ist für unsere Belange weniger kritisch; wir binden sie als nützliches Beispiel für die generische Programmierung in Java ein.

Um mit konkreten Typen von Elementen und Schlüsseln zu arbeiten, deklarieren wir alle relevanten Operationen auf ihnen in einer expliziten Schnittstelle und stellen dann anwendungsspezifische Implementierungen der in der Schnittstelle definierten Operationen bereit. Programm 6.4 ist ein Beispiel für eine derartige Schnittstelle. Neben der obligatorischen `less`-Methode sehen wir auch Methoden vor, um ein zufälliges Element zu generieren, ein Element zu lesen und ein Element in einen String zu konvertieren (damit wir es zum Beispiel ausgeben können).

Programm 6.4 ADT-Schnittstelle für Elemente

Diese ADT-Schnittstelle demonstriert, wie man generische Operationen definiert, die wir auf den zu sortierenden Elementen ausführen wollen: Vergleichen jedes Elements mit einem anderen, Lesen eines Elements von der Standardeingabe, Generieren eines zufälligen Elements und Konvertieren eines Elements in eine Stringdarstellung (um es ausgeben zu können). Jede Implementierung muss `less` einbinden (um die `ITEM`-Schnittstelle zu implementieren) und kann `toString` einbinden (andernfalls wird die Standardroutine von `Object` aufgerufen).

→

```
class myItem implements ITEM // ADT-Schnittstelle
  { // Implementierungen und private Elemente verborgen
    public boolean less(ITEM)
    void read()
    void rand()
    public String toString()
  }
```

Unsere Sortieralgorithmen arbeiten nicht einfach mit Elementen, sondern mit Arrays von Elementen. Dementsprechend ist Programm 6.5 eine ADT-Schnittstelle, die ein *sortierbares Array* abstrahiert. Die in dieser Schnittstelle definierten Operationen beziehen sich auf Arrays – und nicht auf Elemente. Sie sind für Clients vorgesehen, die Sortieralgorithmen testen oder einsetzen. (Wenn ein Client lediglich ein einzelnes Array sortieren muss, lässt es sich leicht einrichten, dass er die Methode `sort` direkt aufrufen kann (siehe Übung 6.11)). Die Methoden in Programm 6.5 stellen nur einige Beispiele von Operationen dar, die sich auf Arrays ausführen lassen. In einer konkreten Anwendung möchten wir gegebenenfalls weitere Operationen definieren (die Klasse `Vector` im Paket `java.util` ist eine Lösung, um eine allgemeine Schnittstelle dieser Art bereitzustellen). Der abstrakte Datentyp von Programm 6.5 konzentriert sich auf sortierbare Arrays. Wie üblich können wir unterschiedliche Implementierungen der verschiedenen Operationen einsetzen, ohne Clientprogramme, die mit dieser Schnittstelle arbeiten, in irgendeiner Form ändern zu müssen.

Programm 6.5 ADT-Schnittstelle für sortierbare Arrays

Die in dieser ADT-Schnittstelle aufgeführten Methoden eignen sich für Clients, die Arrays sortieren: Initialisieren mit zufälligen Werten, Initialisieren mit Werten von der Standardeingabe, Anzeigen des Inhalts und Sortieren des Inhalts. Es ist nicht erforderlich, den Typ der zu sortierenden Elemente anzugeben, um diese Operationen zu definieren.

```
class myArray // ADT-Schnittstelle
  { // Implementierungen und private Elemente verborgen
    myArray(int)
    void rand()
    void read()
    void show(int, int)
    void sort(int, int)
  }
```

Zum Beispiel ist Programm 6.6 ein einfaches Testprogramm, das die gleiche allgemeine Funktionalität wie das Hauptprogramm in Programm 6.1 erfüllt. Es liest entweder ein Array von der Standardeingabe oder generiert zufällige Elemente, sortiert das Array und gibt das Ergebnis aus. Dieses Programm demonstriert, dass wir eine derartige Berechnung ohne Bezug auf den Typ der zu sortierenden Elemente definieren können: Es ist ge-

nerisch. Mit demselben ADT können wir Testprogramme schreiben, die kompliziertere Aufgaben durchführen, und sie dann für Arrays verschiedener Elementtypen verwenden, ohne den Implementierungscode zu ändern (siehe Übung 6.12).

Programm 6.6
Sortiertestprogramm für sortierbare Arrays

Dieses Testprogramm für das Sortieren von Arrays füllt ein generisches Array mit generischen Elementen, sortiert es und zeigt das Ergebnis an. Das Programm interpretiert das erste Befehlszeilenargument als Anzahl der zu sortierenden Elemente und das Vorhandensein eines zweiten Arguments als Kennzeichen dafür, ob zufällige Elemente generiert oder vom Standardeingabestrom gelesen werden sollen. Da das Programm den abstrakten Datentyp von Programm 6.5 verwendet, bezieht sich der Code nicht auf den Typ der zu sortierenden Elemente.

Durch diese Anordnung können wir nicht nur jede Sortierimplementierung zum Sortieren verschiedener Typen von Daten verwenden, ohne irgendwelchen Code zu ändern, sondern auch Methoden für Arrays unabhängig entwickeln (z.B. um unterschiedliche Testprogramme zu schreiben).

```
class ArraySort
  {
    public static void main(String[] args)
      { int N = Integer.parseInt(args[0]);
        myArray A = new myArray(N);
        if (args.length < 2) A.rand(); else A.read();
        A.sort(0, N-1);
        A.show(0, N-1);
      }
  }
```

Programm 6.7 ist eine Implementierung der ADT-Schnittstelle für sortierbare Arrays von Programm 6.5 als Client des generischen Element-ADTs von Programm 6.4. Um ein Array von der Standardeingabe zu lesen, lesen wir Elemente; um ein zufälliges Array zu generieren, generieren wir zufällige Elemente; um ein Array auszugeben, geben wir seine Elemente aus, und um ein Array zu sortieren, verwenden wir `Sort.sort`. Durch die modulare Organisation können wir je nach Anwendung andere Implementierungen einsetzen, beispielsweise eine Implementierung, bei der die Methode `show` nur einen Teil des Arrays ausgibt, wenn man Sortieroperationen auf sehr großen Arrays testet.

Programm 6.7
Beispielimplementierung eines ADTs für sortierbare Arrays

Diese Implementierung verwendet die generische Elementschnittstelle von Programm 6.4, um ein privates Array von Elementen zu verwalten, und verwendet die Methoden `rand`, `read` und `toString` für Elemente, um die entsprechenden Methoden für Arrays zu implementieren. Um flexibler zu sein, behalten wir die Methode `sort` in einer statischen Klasse `Sort`. Implementierungen von `sort` verwenden die Methode `less` aus der `myItem`-Implementierung.

```
class myArray
  {
    private myItem[] a;
    private int N;
    myArray(int N)
      {
        this.N = N;
        a = new myItem[N];
        for (int i = 0; i < N; i++)
          a[i] = new myItem();
      }
    void rand()
      { for (int i = 0; i < N; i++) a[i].rand(); }
    void read()
      { for (int i = 0; i < N; i++)
          if (!In.empty()) a[i].read(); }
    void show(int l, int r)
      { for (int i = l; i <= r; i++)
          Out.println(a[i] + ""); }
    void sort(int l, int r)
      { Sort.sort(a, l, r); }
  }
```

Schließlich betrachten wir ADT-Implementierungen, die wir für verschiedene Typen von Elementen benötigen (der Sinn der Übung). Zum Beispiel gibt Programm 6.8 eine Implementierung des `myItem`-ADTs an, mit der wir Programm 6.6 Arrays von Ganzzahlen sortieren lassen können. Diese Implementierung lässt sich mit jedem Sortierclient, jeder Sortierimplementierung und jeder Arrayimplementierung einsetzen, ohne dass man irgendwelchen anderen Client- oder Implementierungscode ändern müsste.

Programm 6.8 ADT-Implementierung für ganzzahlige Elemente

Dieser Code implementiert die generische `myItem`-ADT-Schnittstelle nach Programm 6.4 für Datensätze, die ganzzahlige Schlüssel sind.

```
class myItem implements ITEM
  { private int key;
    public boolean less(ITEM w)
      { return key < ((myItem) w).key; }
    void read()
      { key = In.getInt(); }
    void rand()
      { key = (int) (1000 * Math.random()); }
    public String toString() { return key + ""; }
  }
```

Für andere Arten von Datensätzen und Schlüsseln lassen sich in unkomplizierter Weise Implementierungen ähnlich Programm 6.8 entwickeln, sodass dieser Mechanismus unseren Sortierimplementierungen ein breites Anwendungsspektrum eröffnet. Für einfache Typen erhalten wir diese Flexibilität zum üblichen Preis einer zusätzlichen Ebene der Indirektion, worauf wir später in diesem Abschnitt zu sprechen kommen. Allerdings arbeiten wir in den meisten praktischen Fällen nicht mit einfachen Typen, sondern mit Datensätzen, in denen alle Arten von Informationen mit den Schlüsseln verbunden sind – hier machen die Vorteile eines generischen Elementtyps die Kosten mehrfach wett. Dieser Abschnitt schließt mit zwei weiteren Beispielen ab, die zeigen, wie einfach sich solche Anwendungen handhaben lassen.

Wir betrachten eine Buchhaltungsanwendung, die als Schlüssel die Kontonummer eines Kunden verwendet sowie einen String mit dem Namen des Kunden und eine Gleitkommazahl für den Kontostand speichert. Programm 6.9 zeigt die Implementierung einer Klasse für solche Datensätze. Nehmen wir nun an, dass wir die Datensätze in sortierter Reihenfolge verarbeiten möchten. Einmal brauchen wir sie alphabetisch nach dem Namen sortiert, ein anderes Mal möchten wir sie in der Reihenfolge der Kontonummern oder der Kontostände anzeigen. Programm 6.10 zeigt, wie sich diese Sortierungen einrichten lassen. Dazu leitet es eine Klasse von `Record` ab, die die Schnittstelle `ITEM` implementiert und eine Methode einbindet, mit der sich der Sortierschlüssel spezifizieren lässt.

Programm 6.9 Beispiel einer Datensatzklasse

Dieses Beispiel veranschaulicht eine typische Klasse für Datensätze in einer Datenverarbeitungsanwendung. Die Klasse hat drei Felder: einen String, eine Ganzzahl und eine Gleitkommazahl. Beispielsweise kann man in diesen Feldern einen Kundennamen, eine Kontonummer bzw. einen Kontostand speichern.

→

```
class Record
  {
    int id;
    double balance;
    String who;
    static int SortKeyField = 0;
    public String toString()
      { return id + " " + balance + " " + who; }
  }
```

Ist beispielsweise Programm 6.10 die Datei `myItem.java` (vielleicht in einem Verzeichnis, das jeweils durch den Klassenpfad spezifiziert wird), dann sortiert Programm 6.6 Datensätze nach dem Kontonummernfeld. Sollen die Datensätze in der Reihenfolge der Kontostände erscheinen, setzen wir einfach das Sortierschlüsselfeld (`SortKeyField`) auf 1:

```
Record.SortKeyField = 1;
```

Wenn wir das Sortierschlüsselfeld auf 2 setzen, erhalten wir eine Sortierung nach dem Namensfeld. Es ist auch möglich, `sort` mehrfach mit jeweils anderen Werten für das Sortierschlüsselfeld aufzurufen, um die Datensätze nach unterschiedlichen Schlüsseln zu sortieren. Um diesen Mechanismus zu implementieren und aufzurufen, sind keinerlei Änderungen im Sortiercode selbst erforderlich.

Programm 6.10
ADT-Implementierung für Datensatzelemente

Dieses Beispiel zeigt, wie wir die `ITEM`-Schnittstelle und den `myItem`-ADT implementieren können, indem wir eine andere Klasse – in diesem Fall die Datensatzklasse von Programm 6.9 – erweitern. Nicht angegeben ist die Implementierung der Methode `rand`. Durch die Implementierung von `less` können Clients das für das Sortieren verwendete Feld ändern.

```
class myItem extends Record implements ITEM
  {
    public boolean less(ITEM w)
      { myItem r = (myItem) w;
        switch (SortKeyField)
          {
            case 2: return who.compareTo(r.who) < 0;
            case 1: return balance < r.balance;
            default: return id < r.id;
          }
      }
    void read()
      {
        id = In.getInt();
        balance = In.getDouble();
        who = In.getString();
      }
  }
```

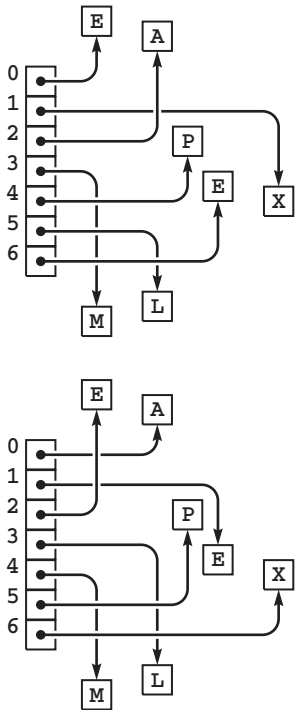


Abbildung 6.2: Zeigersortieren

In Java sortieren wir normalerweise eine Gruppe von Objekten, indem wir Referenzen (Zeiger) auf sie umordnen und nicht die Objekte selbst verschieben. Die obere Zeichnung zeigt ein typisches zu sortierendes Array, das Referenzen auf Objekte mit den Schlüsseln E X A M P L E *in dieser Reihenfolge enthält. Die untere Zeichnung stellt das Ergebnis einer Sortierung dar, wobei die Referenzen im Array auf die Objekte in der Reihenfolge* A E E L M P X *verweisen.*

Die in Programm 6.10 veranschaulichte allgemeine Lösung, eine Klasse, die die Schnittstelle `ITEM` implementiert, abzuleiten, ist in vielen Anwendungen nützlich. Man kann sich ein umfangreiches Softwarepaket vorstellen, das auf der Verarbeitung solcher Datensätze beruht; mit dieser Lösung können wir es ohne großen Zusatzaufwand und ohne Änderungen an bestehender Software mit Sortierfunktionalität ausstatten.

Nach dem gleichen Verfahren können wir unsere Sortierroutinen für alle Arten von Daten einrichten – beispielsweise komplexe Zahlen (siehe Übung 6.13), Vektoren (siehe Übung 6.18) oder Polynome (siehe Übung 6.19) – und brauchen den Sortiercode überhaupt nicht zu ändern. Zwar verlangen kompliziertere Elementtypen auch nach komplizierteren ADT-Implementierungen, doch ist diese Implementierungsarbeit komplett getrennt von den Fragen des Algorithmenentwurfs. Die gleichen Mechanismen können wir mit den meisten Sortierverfahren, die wir in diesem Kapitel betrachten, und ebenso mit denen, die wir in den Kapiteln 7 bis 9 behandeln, einsetzen. In Abschnitt 6.10 sehen wir uns eine wichtige Ausnahme an – sie führt zu einer ganzen Familie wichtiger Sortieralgorithmen, die einen anderen Rahmen erfordern und Thema von Kapitel 10 sind.

Das eben beschriebene Verfahren bezeichnet man in der klassischen Literatur als *Zeiger*sortieren, weil wir Referenzen auf Elemente verarbeiten und nicht die Daten selbst bewegen. In Programmiersprachen wie C und C++ kann der Programmierer explizit Zeiger manipulieren; in Java geschieht die Zeigermanipulierung implizit. Außer bei einfachen numerischen Typen manipulieren wir *immer* Referenzen auf Objekte (Zeiger) und nicht die Objekte an sich.

Sehen wir uns zum Beispiel an, wie sich das Sortieren von Zeichen (indem wir etwa alle `double`-Typen in Programm 6.1 in `char` ändern) und die Verwendung einer `myItem`-Implementierung wie Programm 6.8 (nur mit einem `char`-Schlüssel) unterscheiden (siehe Abbildung 6.2). Im ersten Fall tauschen wir die Zeichen selbst aus und legen sie im Array geordnet ab; im zweiten Fall vertauschen wir Referenzen auf `myI-em`-Objekte, die die Zeichenwerte enthalten.

Wenn wir nichts weiter tun, als eine riesige Datei mit Zeichen zu sortieren, tragen wir die Kosten für eine ebenso große Anzahl von Referenzen plus die Zusatzkosten für den Zugriff auf die Zeichen über die Referenzen. Falls wir obendrein ein `Character`-Objekt als Schlüsselfeld in der `myItem`-Implementierung verwenden, kommt noch eine weitere Ebene der Referenzen hinzu.

Bei großen Datensätzen, die man normalerweise in praktischen Anwendungen antrifft, ist der für die Referenzen erforderliche Platz klein verglichen mit dem Platzbedarf der Datensätze selbst. Die zusätzlich erforderliche Zeit, um den Referenzen zu folgen, wird durch Zeiteinsparungen mehr als kompensiert, weil die Datensätze nicht verschoben werden müssen. Andererseits bietet Java für kleine Datensätze keine gute Lösung, außer dass man Datensätze in einem einfachen numerischen Typ kodiert – ein sicher nur für Experten praktikables Vorgehen.

Als Nächstes wenden wir uns dem Problem zu, Strings zu sortieren. Programm 6.11 veranschaulicht die natürlichste Vorgehensweise in Java: eine direkte Implementierung des abstrakten Datentyps `myItem` für `String`-Objekte mit einem `String`-Feld für den Schlüssel. Diese Implementierung (die für jede Art von Objekt funktioniert) führt eine Ebene von Referenzen ein: das Array enthält Referenzen auf `myItem`-Objekte, die `String`-Objekte enthalten, die ihrerseits Referenzen auf Folgen von Zeichen sind. Es gibt verschiedene Wege, um diese zweite Ebene der Indirektion in Java bei Bedarf zu umgehen, jedoch ist es für Programmierer in Java üblich, zusätzliche Ebenen der Indirektion zugunsten eines leicht verständlichen Referenzmodells zu akzeptieren. Da Java eine automatische Speicherverwaltung besitzt, fallen von vornherein viele Probleme weg, mit denen Programmierer in anderen Sprachen zu kämpfen haben.

Programm 6.11
ADT-Implementierung für Stringelemente

Dieser Code implementiert den generischen `myItem`-ADT von Programm 6.4 für Datensätze, die Stringschlüssel sind.

```
class myItem implements ITEM
  { String key;
    public boolean less(ITEM w)
      { return key.compareTo(((myItem) w).key)<0; }
    void read()
      { key = In.getString(); }
    void rand()
      { int a = (int)('a'); key = "";
        for (int i = 0; i < 1+9*Math.random(); i++)
          key += (char) (a + 26*Math.random());
      }
    public String toString() { return key; }
  }
```

Mit derartigen Fragen der Speicherverwaltung haben wir immer dann zu tun, wenn wir ein Programm in Module aufteilen. Wer sollte für die Verwaltung des Speichers entsprechend der konkreten Realisierung eines Objekts verantwortlich sein: der Client, die Datentypimplementierung oder das System? Wer entscheidet, welcher Speicher ungenutzt ist, wer gibt ihn in diesem Fall wieder frei? In vielen Sprachen gibt es für derartige Fragen keine bindenden Richtlinien; in Java übernimmt das System die Verantwortlichkeit.

Das Java-Konzept bietet sicherlich viele Vorteile. Im Rahmen der Sortierverfahren lässt sich mithilfe der Referenzen vermeiden, dass man die zu sortierenden Daten antastet. Wir können eine Datei selbst dann »sortieren«, wenn sie lediglich Lesezugriffe erlaubt. Darüber hinaus lassen sich bei Arrays mit mehreren Referenzen zwei unterschiedlich sortierte Darstellungen ein und derselben Datenmenge erzeugen. Diese Flexibilität, die Daten zu manipulieren ohne sie tatsächlich zu ändern, ist in vielen Anwendungen sehr nützlich. Eine umfassende Behandlung aller Fragen geht über den Rahmen dieses Buchs hinaus, wir müssen aber den Einfluss auf die Leistung berücksichtigen, damit wir die kritischen Stellen in Anwendungen verbessern können. In Abschnitt 6.6 kommen wir auf diese Fragen zurück.

Mit Referenzen lassen sich außerdem die Kosten dafür vermeiden, vollständige Datensätze zu verschieben. Die Kosteneinsparungen sind für Daten mit großen Datensätzen (und kleinen Schlüsseln) beträchtlich, weil die Vergleichsoperationen nur auf einen kleinen Teil des Datensatzes zugreifen müssen und der größte Teil des Datensatzes beim Sortieren nicht einmal berührt wird. Das Referenzkonzept macht die Kosten für das Vertauschen ungefähr gleich den Kosten eines Vergleichs für allgemeine Situationen mit beliebig langen Datensätzen (zum Preis des zusätzlichen Platzes für die Referenzen). Bei langen Schlüsseln können die Vertauschungen sogar billiger sein als die Vergleiche. Wenn wir die Laufzeiten von Methoden einschätzen, die Dateien mit Ganzzahlen sortieren, gehen wir oftmals davon aus, dass sich die Kosten der Vergleiche und Vertauschungen nicht wesentlich unterscheiden. Schlussfolgerungen, die auf dieser Annahme fußen, gelten wahrscheinlich für eine weite Klasse von Anwendungen, wenn wir Referenzobjekte sortieren.

Der in diesem Abschnitt diskutierte Ansatz bildet einen Mittelweg zwischen Programm 6.1 und einer industrietauglichen, vollständig abstrakten Menge von Implementierungen, die mit Fehlerprüfungen, externer Speicherverwaltung und noch allgemeineren Fähigkeiten ausgestattet sind. Derartige Rahmenbedingungen werden in modernen Programmier- und Anwendungsumgebungen zunehmend wichtiger. Einige Fragen müssen wir unbeantwortet lassen. Über den relativ einfachen Mechanismus, den wir untersucht haben, wollen wir vor allem demonstrieren, dass die von uns behandelten Sortierimplementierungen ein breites Anwendungsspektrum haben.

ÜBUNGEN

▷ 6.10 Schreiben Sie eine `myItem`-ADT-Implementierung für das Sortieren von `double`-Werten.

▷ 6.11 Nehmen Sie an, dass Clients lediglich ein einzelnes Array von Objekten des Typs `String` sortieren müssen. Schreiben Sie für diesen Zweck eine Implementierung wie Programm 6.3 und beschreiben Sie, wie sie die Clients verwenden sollen.

6.12 Modifizieren Sie Ihre Testprogramme der Übungen 6.8 und 6.9, um `Sort.sort` zu verwenden. Ergänzen Sie Programm 6.3 durch öffentliche Methoden, um die Anzahl der Vergleiche und Vertauschungen der zuletzt ausgeführten Sortieroperation zurückzugeben, damit Sie diese Kenngrößen zusätzlich zur Laufzeit studieren können.

6.13 Schreiben Sie Klassen, mit denen Sortiermethoden komplexe Zahlen $x + iy$ mit dem Absolutbetrag $\sqrt{x^2 + y^2}$ für den Schlüssel sortieren können. *Hinweis:* Wenn man die Quadratwurzel ignoriert, lässt sich die Effizienz verbessern.

▷ 6.14 Ergänzen Sie den Array-ADT in Programm 6.5 durch eine Methode `check` und geben Sie eine Implementierung für Programm 6.7 an, die `true` zurückgibt, wenn das Array in sortierter Reihenfolge vorliegt, und andernfalls `false` liefert.

○ 6.15 Geben Sie eine Implementierung der Methode `check` aus Übung 6.14 an, die `true` zurückgibt, wenn das Array sortiert vorliegt *und* die Menge der Elemente im Array gleich der Menge der Elemente ist, die beim Aufruf von `sort` im Array enthalten war, und andernfalls `false` liefert.

○ 6.16 Entwerfen und implementieren Sie eine Testmöglichkeit im Geist der Übungen 6.14 und 6.15, die `true` zurückgibt, wenn der letzte Aufruf von `sort` stabil war, und andernfalls `false` liefert.

• 6.17 Erweitern Sie die Methode `rand` in Ihrer Implementierung von `myItem` für `double`-Werte (siehe Übung 6.10), sodass sie Testdaten ähnlich den in Abbildung 6.15 dargestellten Verteilungen generiert. Sehen Sie einen Ganzzahlparameter vor, mit dem der Client die Verteilung spezifizieren kann.

○ 6.18 Schreiben Sie Klassen, mit denen Sortiermethoden mehrdimensionale Vektoren von d Ganzzahlen sortieren können. Die Vektoren sind zunächst nach der ersten Komponente zu sortieren, innerhalb gleicher erster Komponenten nach der zweiten Komponente, Vektoren mit gleichen ersten und zweiten Komponenten nach der dritten Komponente usw.

6.19 Schreiben Sie Klassen, mit denen Sortiermethoden Polynome sortieren können (siehe Abschnitt 4.10). Ein Teil Ihrer Aufgabe besteht darin, ein geeignetes Ordnungsschema zu definieren.

6.3 Sortieren durch Auswählen

Einer der einfachsten Sortieralgorithmen arbeitet wie folgt: Suche zuerst das kleinste Element im Array und tausche es dann mit dem Element an der ersten Position aus. Suche dann das zweitkleinste Element und tausche es mit dem Element an der zweiten Position aus. Fahre auf diese Weise fort, bis das gesamte Array sortiert ist. Dieses Verfahren heißt *Sortieren durch Auswählen* (Selection Sort), weil es wiederholt das kleinste verbleibende Element auswählt. Abbildung 6.3 zeigt den laufenden Sortiervorgang an einer Beispieldatei.

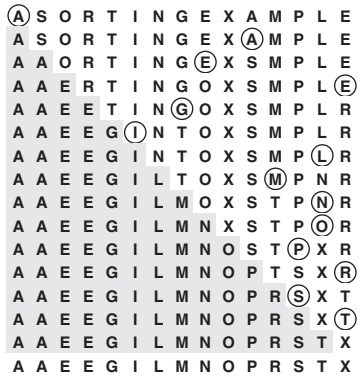


Abbildung 6.3: Beispiel für Sortieren durch Auswählen

Der erste Durchgang hat in diesem Beispiel keine Wirkung, weil das Array kein Element enthält, das kleiner als das links stehende A *ist. Im zweiten Durchgang ist das andere* A *das kleinste verbliebene Element, sodass es mit* S *an der zweiten Position ausgetauscht wird. Dann wird das* E *in der Mitte mit dem* O *an der dritten Position getauscht, anschließend – im vierten Durchgang – das andere* E *mit dem* R *an der vierten Position usw.*

Programm 6.12 ist eine Implementierung für das Sortieren durch Auswählen gemäß unseren Konventionen. Die innere Schleife enthält lediglich einen Vergleich, um das aktuelle Element gegen das bisher kleinste gefundene Element zu testen (dazu kommt noch der Code, um den Index des aktuellen Elements zu inkrementieren und zu prüfen, ob er die Arraygrenzen nicht überschreitet); es könnte kaum einfacher sein. Die Umspeicherung der Elemente fällt aus der inneren Schleife heraus: Jedes Vertauschen legt ein Element an seiner endgültigen Position ab, sodass die Anzahl der Vertauschungen gleich $N - 1$ ist (für das letzte Element ist kein Vertauschen erforderlich). Die Laufzeit wird somit durch die Anzahl der Vergleiche dominiert. In Abschnitt 6.6 zeigen wir, dass diese Anzahl proportional zu N^2 ist. Außerdem untersuchen wir genauer, wie sich die Gesamtlaufzeit bestimmen lässt und wie man das Sortieren durch Auswählen mit anderen elementaren Sortierverfahren vergleicht.

Programm 6.12 Sortieren durch Auswählen

Für jedes `i` von `l` bis `r–1` tauscht das Programm `a[i]` mit dem kleinsten Element in `a[i], ..., a[r]` aus. Während der Index `i` von links nach rechts wandert, stehen die Elemente links vom Index an ihrer endgültigen Position im Array (und werden nicht mehr »angefasst«), sodass das Array vollständig sortiert ist, wenn `i` das rechte Ende erreicht.

→

Wie Abschnitt 6.2 detailliert erläutert hat, kennzeichnen wir in diesem Code den Typ des zu sortierenden Elements mit dem Schlüsselwort `ITEM`. Um den Code in Programm 6.1 zu verwenden, ersetzen wir `ITEM` durch `double`; genauso gehen wir vor, um Arrays mit Elementen einfacher Typen zu sortieren. Um diesen Code für Referenztypen einzusetzen, implementieren wir die `ITEM`-Schnittstelle von Programm 6.2, ersetzen die Methode `example` in Programm 6.3 durch diese Methode und ändern die Aufrufe von `example` in Aufrufe dieser Methode. Die gleiche Konvention verwenden wir für alle unsere Implementierungen von Sortieralgorithmen.

```
static void selection(ITEM[] a, int l, int r)
  {
    for (int i = l; i < r; i++)
      { int min = i;
        for (int j = i+1; j <= r; j++)
          if (less(a[j], a[min])) min = j;
        exch(a, i, min);
      }
  }
```

Ein Nachteil beim Sortieren durch Auswählen besteht darin, dass die Laufzeit des Algorithmus in gewissem Maße vom Umfang der bereits in der Datei vorliegenden Sortierung abhängt. Sucht man in einem Durchgang durch die Datei das kleinste Element, erhält man offenbar kaum Informationen darüber, wo das Minimum beim nächsten Durchgang liegen könnte. Beispielsweise könnte der Benutzer dieses Sortierverfahrens überrascht sein, dass der Algorithmus zum Sortieren durch Auswählen für eine bereits sortierte Datei oder für eine Datei, in der alle Schlüssel gleich sind, genauso lange läuft wie für eine zufällig geordnete Datei! Wir werden noch sehen, dass andere Verfahren eine vorliegende Ordnung in der Eingabedatei besser nutzen können.

Trotz seiner Einfachheit und des offensichtlichen Brute-Force-Ansatzes stellt das Sortieren durch Auswählen ausgeklügeltere Verfahren bei einer wichtigen Anwendung in den Schatten: Es ist das bevorzugte Verfahren, um Dateien mit sehr großen Elementen und kleinen Schlüsseln zu sortieren. Bei solchen Anwendungen sind die Kosten für das Umordnen der Daten weit höher als die Kosten für die Vergleichsoperationen – und kein Algorithmus kann eine Datei mit wesentlich weniger Datenverschiebungen sortieren als das Sortieren durch Auswählen (siehe Eigenschaft 6.5 in Abschnitt 6.6).

ÜBUNGEN

▷ 6.20 Zeigen Sie in der Art von Abbildung 6.3, wie das Sortieren durch Auswählen die Beispieldatei `E A S Y Q U E S T I O N` sortiert.

6.21 Wie groß ist die maximale Anzahl von Vertauschungen für ein bestimmtes Element beim Sortieren durch Auswählen? Wie groß ist die durchschnittliche Anzahl der Vertauschungen eines Elements?

➔ 6.22 Geben Sie ein Beispiel für eine Datei von N Elementen an, die beim Sortieren durch Auswählen die höchste Anzahl von gescheiterten Tests `a[j] < a[min]` liefert (wobei folglich `min` aktualisiert wird).

○ 6.23 Ist das Sortieren durch Auswählen ein stabiler Algorithmus?

6.4 Sortieren durch Einfügen

Wenn ein Skatspieler seine Handkarten sortiert, geht er in der Regel so vor, dass er sich jeweils eine Karte ansieht und sie an die richtige Position in den bereits betrachteten Karten steckt (und das Blatt damit sortiert hält). In einer Computerimplementierung müssen wir Platz für das einzufügende Element schaffen, indem wir die größeren Elemente um eine Position nach rechts schieben und das Element an der freien Position einfügen. Die Methode `sort` in Programm 6.1 ist eine Implementierung dieses Verfahrens, das man als *Sortieren durch Einfügen* (Insertion Sort) bezeichnet.

Wie beim Sortieren durch Auswählen sind die Elemente während des Sortiervorgangs links vom aktuellen Index sortiert, befinden sich aber noch nicht an ihrer endgültigen Position, da sie gegebenenfalls verschoben werden müssen, um Platz für später angetroffene kleinere Elemente zu schaffen. Dennoch ist das Array vollständig sortiert, wenn der Index das rechte Ende erreicht. Abbildung 6.4 zeigt das laufende Verfahren an einer Beispieldatei.

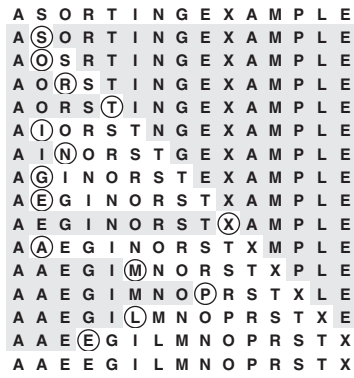


Abbildung 6.4: Beispiel für das Sortieren durch Einfügen

Im ersten Durchgang beim Sortieren durch Einfügen ist das S *an der zweiten Position größer als das* A*, sodass es nicht verschoben werden muss. Wenn der zweite Durchgang auf das* O *an der dritten Position trifft, wird es mit dem* S *getauscht, um* A O S *in die richtige Reihenfolge zu bringen, usw. Nicht schattierte Elemente ohne Kreis wurden lediglich um eine Position nach rechts geschoben.*

Die Implementierung des Verfahrens *Sortieren durch Einfügen* in Programm 6.1 ist unkompliziert, aber ineffizient. Wir sehen uns jetzt drei Möglichkeiten an, mit denen sich das Verfahren verbessern lässt, um ein wiederkehrendes Schema zu veranschaulichen, das sich durch viele unserer Implementierungen zieht: Wir möchten, dass der Code prägnant, klar und effizient ist, wobei diese Ziele oftmals im Widerspruch stehen, sodass wir meistens einen Kompromiss eingehen müssen. Dazu entwickeln wir eine natürliche Imple-

mentierung, versuchen dann, sie mithilfe einer Reihe von Transformationen zu verbessern und prüfen die Wirksamkeit (und Korrektheit) jeder Transformation.

Programm 6.13 ist eine Implementierung für das Sortieren durch Einfügen, die effizienter als die in Programm 6.1 angegebene Version ist (in Abschnitt 6.6 zeigen wir, dass sie nahezu doppelt so schnell arbeitet). In diesem Buch sind wir *sowohl* an eleganten und effizienten Algorithmen *als auch* an ihren eleganten und effizienten Implementierungen interessiert. Im vorliegenden Fall unterschieden sich die zugrunde liegenden Algorithmen leicht – bei der `sort`-Methode von Programm 6.1 sollten wir besser von einem *nichtadaptiven Sortieren durch Einfügen* sprechen. Wenn man die Eigenschaften eines Algorithmus verstanden hat, ist es auch leichter, eine Implementierung zu entwickeln, die sich effizient in einer Anwendung nutzen lässt.

Programm 6.13 Sortieren durch Einfügen

Dieser Code weist gegenüber der Implementierung von `sort` in Programm 6.1 folgende Verbesserungen auf: (1.) Er bringt zuerst das kleinste Element im Array auf die erste Position, sodass dieses Element als Markierungsschlüssel dienen kann; (2.) in der inneren Schleife steht nur eine einzige Zuweisung und keine Austauschoperation; (3.) die innere Schleife terminiert, wenn sich das einzufügende Element an seiner Position befindet. Für jedes `i` sortiert das Programm die Elemente `a[l], ..., a[i]`, indem es die Elemente, die in der sortierten Liste `a[l], ..., a[i-1]` größer als `a[i]` sind, um eine Position nach rechts verschiebt und dann `a[i]` an seine richtige Position bringt.

```
static void insertion(ITEM[] a, int l, int r)
  { int i;
    for (i = r; i > l; i--) compExch(a, i-1, i);
    for (i = l+2; i <= r; i++)
      { int j = i; ITEM v = a[i];
        while (less(v, a[j-1]))
          { a[j] = a[j-1]; j--; }
        a[j] = v;
      }
  }
```

Erstens können wir mit den Operationen von `compExch` aufhören, wenn wir auf einen Schlüssel treffen, der nicht größer ist als der Schlüssel im einzufügenden Element, weil das Teilarray links vom Index sortiert ist. Insbesondere können wir die innere `for`-Schleife in Programm 6.1 vorzeitig durch eine `break`-Anweisung verlassen, wenn die Bedingung `a[j-1] < a[j]` erfüllt (`true`) ist. Diese Modifikation überführt die Implementierung in eine adaptive Sortierroutine und beschleunigt das Programm um einen Faktor von etwa 2 bei zufällig angeordneten Schlüsseln (siehe Eigenschaft 6.2).

Bei der eben beschriebenen Verbesserung haben wir zwei Bedingungen, die die innere Schleife beenden – wir könnten sie als `while`-Schleife formulieren, um diese Tatsache ex-

plizit widerzuspiegeln. Eine fast unauffällige Verbesserung der Implementierung folgt aus der Anmerkung, dass der Test `j > l` gewöhnlich überflüssig ist; in der Tat verläuft er *nur* erfolgreich, wenn das einzufügende Element das bisher kleinste besuchte ist und den Anfang des Arrays erreicht. Eine gebräuchliche Alternative ist, die zu sortierenden Schlüssel in `a[1]` bis `a[N]` zu halten und in `a[0]` einen *Markierungsschlüssel* zu schreiben, der mindestens so klein wie das kleinste Element im Array ist. Wenn das Programm dann prüft, ob ein kleinerer Schlüssel angetroffen wurde, testet es gleichzeitig beide infrage kommenden Bedingungen, wodurch die innere Schleife kürzer und das Programm schneller wird.

Markierungsschlüssel sind manchmal unhandlich: Vielleicht lässt sich der kleinste mögliche Schlüssel nicht ohne weiteres angeben oder die aufrufende Routine bietet keinen Platz, um einen zusätzlichen Schlüssel einzubinden. Programm 6.13 veranschaulicht einen Weg, um diese beiden Probleme für das Sortieren durch Einfügen zu umgehen: Wir führen einen expliziten ersten Durchgang über das Array durch, der das Element mit dem kleinsten Schlüssel an die erste Position stellt. Dann sortieren wir das restliche Array, wobei jetzt das erste und kleinste Element als Markierungsschlüssel dient. Im Allgemeinen sollte man auf Markierungsschlüssel im Code verzichten, weil Code mit expliziten Tests oftmals verständlicher ist. In vielen Situationen ist aber ein Markierungsschlüssel nützlich, um Programme sowohl einfacher als auch effizienter zu machen.

Als dritte Verbesserung können wir überflüssige Anweisungen aus der inneren Schleife entfernen. Es folgt aus der Anmerkung, dass aufeinander folgende Vertauschungen, an denen dasselbe Element beteiligt ist, ineffizient sind. Wenn es zwei oder mehr Vertauschungen gibt, haben wir

```
t = a[j]; a[j] = a[j-1]; a[j-1] = t;
```

gefolgt von

```
t = a[j-1]; a[j-1] = a[j-2]; a[j-2] = t;
```

usw. Der Wert von `t` ändert sich zwischen diesen beiden Sequenzen nicht und wir verschwenden nur Zeit, wenn wir ihn speichern und dann für die nächste Vertauschung erneut laden. Programm 6.13 verschiebt größere Elemente um eine Position nach rechts, anstatt Vertauschungen vorzunehmen und auf diese Weise Zeit zu verschwenden.

Im Gegensatz zum Sortieren durch Auswählen hängt die Laufzeit beim Sortieren durch Einfügen hauptsächlich von der anfänglichen Reihenfolge der Schlüssel in den Eingabedaten ab. Wenn zum Beispiel die Datei groß ist und die Schlüssel schon sortiert (oder fast sortiert) vorliegen, dann ist Sortieren durch Einfügen schnell und Sortieren durch Auswählen langsam. In Abschnitt 6.6 vergleichen wir die Algorithmen im Detail.

ÜBUNGEN

▷ 6.24 Zeigen Sie in der Art von Abbildung 6.4, wie das Sortieren durch Einfügen die Beispieldatei `E A S Y Q U E S T I O N` sortiert.

→

→

6.25 Geben Sie eine Implementierung für das Sortieren durch Einfügen an, bei der die innere Schleife als `while`-Schleife kodiert ist, die wie im Text beschrieben bei einer von zwei Bedingungen terminiert.

6.26 Beschreiben Sie für jede der Bedingungen in der `while`-Schleife nach Übung 6.25 eine Datei von N Elementen, bei der die Bedingung immer falsch ist, wenn die Schleife terminiert.

○ 6.27 Ist das Sortieren durch Einfügen stabil?

6.28 Geben Sie eine nichtadaptive Implementierung für das Sortieren durch Auswählen an, die darauf basiert, das kleinste Element mit Code wie für die erste `for`-Schleife in Programm 6.13 zu suchen.

6.5 Bubblesort

Die meisten angehenden Programmierer lernen als erstes Sortierverfahren das so genannte *Bubblesort* (Sortieren durch Vertauschen) kennen: Durchlaufe immer wieder die Datei und vertausche benachbarte Elemente, die sich noch nicht in der gewünschten Reihenfolge befinden; fahre so lange fort, bis die Datei sortiert ist. Bubblesort lässt sich vor allem leicht implementieren, wobei es aber fraglich ist, ob es sich tatsächlich leichter als Sortieren durch Einfügen implementieren lässt. Im Allgemeinen läuft Bubblesort langsamer als die beiden anderen Verfahren, der Vollständigkeit halber werfen wir aber einen kurzen Blick darauf.

Angenommen wir durchlaufen die Datei immer von rechts nach links. Wenn während des ersten Durchgangs das kleinste Element angetroffen wird, tauschen wir es mit jedem der Elemente zu seiner Linken aus, sodass es schließlich an das linke Ende des Arrays wandert. Im zweiten Durchgang kommt dann das zweitkleinste Element an seine Position usw. Folglich genügen N Durchläufe, und Bubblesort arbeitet in der Art wie Sortieren durch Auswählen, obwohl es mehr Schritte ausführt, um jedes Element an seine Position zu bringen. Programm 6.14 gibt eine Implementierung an und Abbildung 6.5 zeigt ein Beispiel für den laufenden Algorithmus.

Programm 6.14 Bubblesort

Für jedes `i` von `l` bis `r`–1 bringt die innere Schleife (`j`) das kleinste Element unter den Elementen in `a[i], ..., a[r]` nach `a[i]`, indem das Programm die Elemente von rechts nach links durchläuft, jeweils zwei benachbarte Elemente vergleicht und gegebenenfalls vertauscht. Das kleinste Element bewegt sich bei allen derartigen Vergleichen, sodass es wie eine Blase (engl: bubble) an den Anfang wandert. Wie beim Sortieren durch Auswählen stehen die Elemente links vom Index an ihrer endgültigen Position im Array, während der Index `i` von links nach rechts durch die Datei läuft. →

```
static void bubble(ITEM[] a, int l, int r)
  { for (int i = l; i < r; i++)
      for (int j = r; j > i; j--)
        compExch(a, j-1, j);
  }
```

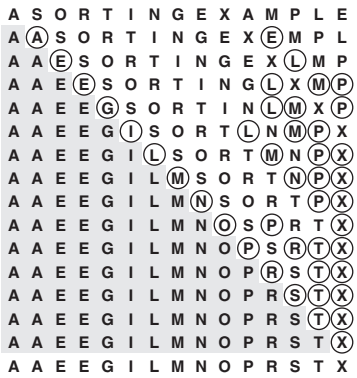


Abbildung 6.5: Beispiel für Bubblesort

In Bubblesort wandern kleine Schlüssel nach links. Der von rechts nach links voranschreitende Sortiervorgang tauscht jeden Schlüssel mit seinem linken Nachbarn aus, bis ein kleinerer Schlüssel gefunden wird. Beim ersten Durchgang wird das E *mit dem* L*, dem* P *und dem* M *ausgetauscht, bevor die Tauschoperationen beim rechten* A *stoppen; dann bewegt sich dieses* A *an den Anfang der Datei und stoppt beim anderen* A*, das sich bereits an seiner endgültigen Position befindet. Der i-kleinste Schlüssel erreicht seine endgültige Position nach dem i-ten Durchgang, genau wie beim Sortieren durch Auswählen, wobei aber auch andere Schlüssel näher an ihre endgültige Position rücken.*

Programm 6.14 lässt sich beschleunigen, wenn man die innere Schleife sorgfältig implementiert, und zwar in der gleichen Art wie in Abschnitt 6.4 beim Sortieren durch Einfügen (siehe Übung 6.34). Vergleicht man den Code, scheint Programm 6.14 identisch mit dem nichtadaptiven Sortieren durch Einfügen in Programm 6.1 zu sein. Der Unterschied zwischen beiden besteht darin, dass die innere `for`-Schleife beim Sortieren durch Einfügen den linken (sortierten) Teil des Arrays durchläuft und bei Bubblesort den rechten (nicht unbedingt sortierten) Teil des Arrays.

Programm 6.14 verwendet nur `compExch`-Anweisungen und ist daher nichtadaptiv. Wir können das Verfahren aber effizienter gestalten, wenn die Datei nahezu geordnet ist, indem wir testen, ob in einem Durchgang überhaupt keine Vertauschungen stattgefunden haben (und sich die Datei folglich in sortierter Reihenfolge befindet, sodass wir die äußere Schleife mit `break` verlassen können). Fügt man diese Verbesserung hinzu, läuft Bubblesort bei bestimmten Arten von Dateien zwar schneller, ist aber im Allgemeinen nicht so effizient, wie es die Änderung zum Verlassen der inneren Schleife beim Sortieren durch Einfügen bewirkt. Darauf geht Abschnitt 6.6 näher ein.

ÜBUNGEN

▷ 6.29 Zeigen Sie in der Art von Abbildung 6.5, wie Bubblesort die Beispieldatei `E A S Y Q U E S T I O N` sortiert.

6.30 Geben Sie ein Beispiel für eine Datei an, bei der die Anzahl der von Bubblesort durchgeführten Vertauschungen maximal ist.

○ 6.31 Ist Bubblesort stabil?

➔ 6.32 Erläutern Sie, warum Bubblesort gegenüber der nichtadaptiven Version des Verfahrens *Sortieren durch Auswählen* von Übung 6.28 vorzuziehen ist.

• 6.33 Ermitteln Sie experimentell für zufällige Dateien mit N Elementen, wie viele Durchläufe sich einsparen lassen, wenn man in Bubblesort einen Test vorsieht, um das Verfahren abzubrechen, wenn die Datei sortiert ist.

6.34 Entwickeln Sie eine effiziente Implementierung von Bubblesort, bei der die innere Schleife möglichst wenig Anweisungen enthält. Vergewissern Sie sich, dass durch Ihre »Verbesserungen« das Programm am Ende nicht langsamer läuft!

6.6 Leistungsdaten elementarer Sortierverfahren

Sortieren durch Auswählen, *Sortieren durch Einfügen* und *Bubblesort* sind Algorithmen mit quadratischer Abhängigkeit der Laufzeit sowohl im ungünstigsten als auch im durchschnittlichen Fall; alle Verfahren kommen ohne zusätzlichen Speicher aus. Ihre Laufzeiten unterscheiden sich daher nur durch einen konstanten Beitrag, sie arbeiten jedoch ziemlich unterschiedlich, wie es die Abbildungen 6.6 bis 6.8 belegen.

Im Allgemeinen ist die Laufzeit eines Sortieralgorithmus proportional zur Anzahl der notwendigen Vergleiche und/oder zur Anzahl der verschobenen oder vertauschten Elemente. Wenn man für zufällige Eingabedaten die Verfahren vergleicht, muss man auch konstante Anteile hinsichtlich Anzahl der Vergleichs- und Austauschoperationen sowie Länge der inneren Schleifen berücksichtigen. Bei Eingabedaten mit speziellen Eigenschaften kann die Laufzeit der Verfahren um mehr als einen konstanten Faktor abweichen. In diesem Abschnitt sehen wir uns die analytischen Ergebnisse genauer an, um diese Schlussfolgerungen zu untermauern.

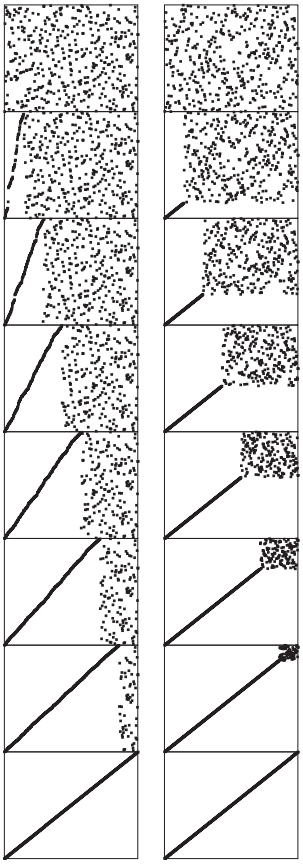


Abbildung 6.6: Dynamische Eigenschaften der Verfahren Sortieren durch Einfügen und Sortieren durch Auswählen

Diese Momentaufnahmen für das Sortieren durch Einfügen (links) und das Sortieren durch Auswählen (rechts) bei laufenden Algorithmen auf einer zufälligen Permutation zeigen, wie jede Methode durch das Sortieren voranschreitet. Ein zu sortierendes Array stellen wir dar, indem wir `i` *über* `a[i]` *für jedes* `i` *auftragen. Vor dem Sortieren sind die Punkte zufällig verteilt; nach dem Sortieren ergibt sich eine diagonal von links unten nach rechts oben verlaufende Linie. Das Sortieren durch Einfügen »schaut« von seiner aktuellen Position im Array nie nach vorn; Sortieren durch Auswählen »blickt nie zurück«.*

Abbildung 6.7: Vergleiche und Vertauschungen in elementaren Sortierverfahren

Diese Zeichnungen heben die Unterschiede in der Art und Weise hervor, nach der die Verfahren Sortieren durch Einfügen, Sortieren durch Auswählen und Bubblesort eine Datei sortieren. Die zu sortierende Datei ist durch Linien dargestellt, die entsprechend ihren Winkeln sortiert werden sollen. Schwarze Linien entsprechen den Elementen, auf die während eines Sortierdurchgangs zugegriffen wird; graue Linien bleiben unberührt. Beim Sortieren durch Einfügen (links) geht das einzufügende Element bei jedem Durchgang etwa zur Hälfte durch den sortierten Teil der Datei zurück. Sortieren durch Auswählen (Mitte) und Bubblesort (rechts) gehen bei jedem Durchgang durch den gesamten nicht sortierten Teil des Arrays, um dort das nächstkleinere Element zu finden; der Unterschied zwischen beiden Verfahren besteht darin, dass Bubblesort jeweils benachbarte Elemente, die nicht der Reihenfolge entsprechen, vertauscht, während Sortieren durch Auswählen nur das kleinste Element auf seine endgültige Position bringt. Aus diesem Unterschied ergibt sich, dass der unsortierte Teil des Arrays bei fortschreitendem Bubblesort immer geordneter wird.

Eigenschaft 6.1 *Sortieren durch Auswählen benötigt ungefähr $N^2/2$ Vergleiche und N Vertauschungen.*

Diese Eigenschaft können wir leicht überprüfen, indem wir den Beispiellauf entsprechend Abbildung 6.3 untersuchen. Die Abbildung stellt eine N-mal-N-Tabelle dar, in der die nicht

schattierten Buchstaben für Vergleiche stehen. Ungefähr die Hälfte der Elemente in der Tabelle sind nicht schattiert – und zwar diejenigen oberhalb der Diagonalen. Die $N - 1$ Elemente auf der Diagonalen (mit Ausnahme des letzten) entsprechen einer Vertauschung. Genauer gesagt ergeben sich aus der Untersuchung des Codes, dass für jedes i von 1 bis $N - 1$ eine Vertauschung und $N - i$ Vergleiche auftreten, insgesamt also $N - 1$ Vertauschungen und $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N - 1)/2$ Vergleiche. Diese Beobachtungen gelten unabhängig von den Eingabedaten; beim Sortieren durch Auswählen besteht die einzige Abhängigkeit von den Eingabedaten darin, wie oft `min` aktualisiert wird. Im ungünstigsten Fall kann diese Größe auch quadratisch sein; im durchschnittlichen Fall ist es lediglich $O(N \log N)$ (siehe den Referenzabschnitt), sodass wir von einer Laufzeit beim Sortieren durch Auswählen ausgehen können, die kaum von den Eingabedaten abhängig ist. ■

Eigenschaft 6.2 *Sortieren durch Einfügen benötigt im durchschnittlichen Fall ungefähr $N^2/4$ Vergleiche und $N^2/4$ Halbvertauschungen (Bewegungen) und doppelt so viel im ungünstigsten Fall.*

Nach der Implementierung in Programm 6.13 ist die Anzahl der Vergleiche und der Bewegungen ungefähr gleich groß. Genau wie bei Eigenschaft 6.1 lässt sich diese Größe leicht in der N-mal-N-Darstellung von Abbildung 6.4 ablesen, die Einzelheiten für die Arbeitsweise des Algorithmus zeigt. Hier werden die Elemente unterhalb der Diagonalen gezählt – im ungünstigsten Fall alle Elemente. Bei zufälligen Eingabedaten ist zu erwarten, dass sich jedes Element im Mittel ungefähr den halben Weg zurückbewegt, sodass unterhalb der Diagonalen die Hälfte der Elemente zu zählen ist. ■

Eigenschaft 6.3 *Bubblesort benötigt im durchschnittlichen und im ungünstigsten Fall ungefähr $N^2/2$ Vergleiche und $N^2/2$ Vertauschungen.*

Der i-te Durchgang von Bubblesort benötigt $N - i$ Vergleichen-Vertauschen-Operationen, sodass der Beweis analog zum Sortieren durch Auswählen erfolgt. Modifiziert man den Algorithmus, sodass er terminiert,

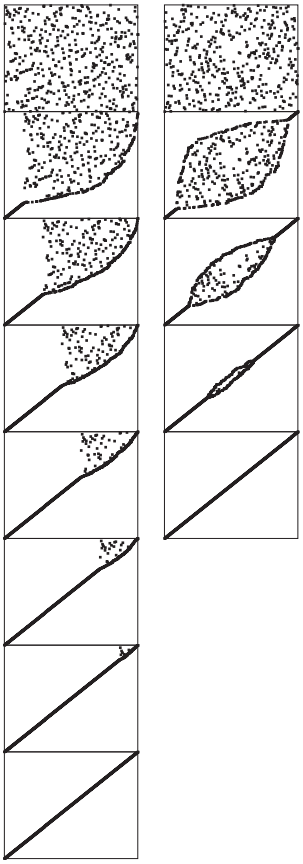


Abbildung 6.8: Dynamische Eigenschaften von zwei Bubblesort-Varianten

Das standardmäßige Bubblesort (links) arbeitet ähnlich dem Sortieren durch Auswählen, indem jeder Durchgang ein Element auf seine Position bewegt, es bringt aber auch in asymmetrischer Weise eine gewisse Ordnung in andere Teile des Arrays. Durchsucht man das Array abwechselnd von Anfang bis Ende und von Ende bis Anfang, erhält man eine Version von Bubblesort, die Shakersort *genannt wird (rechts) und schneller zum Ziel kommt (siehe Übung 6.40).*

wenn er erkennt, dass die Datei sortiert ist, hängt die Laufzeit von den Eingabedaten ab. Es ist nur ein Durchgang erforderlich, wenn die Datei bereits sortiert ist, dagegen erfordert der i-te Durchgang $N - i$ Vergleiche *und* Vertauschungen, wenn die Datei in umgekehrter Reihenfolge vorliegt. Wie bereits erwähnt, ist die Leistung im durchschnittlichen Fall nicht wesentlich besser als im ungünstigsten Fall, auch wenn es recht kompliziert ist, diese Tatsache per Analyse darzustellen (siehe den Referenzabschnitt). ■

Obwohl das Konzept einer teilweise sortierten Datei zwangsläufig ziemlich ungenau ist, arbeiten Sortieren durch Einfügen und Bubblesort besonders gut für bestimmte Arten von nichtzufälligen Dateien, wie sie in der Praxis häufig vorkommen. Für derartige Anwendungen werden oftmals Universalsortierverfahren missbraucht. Sehen Sie sich zum Beispiel die Arbeitsweise von Sortieren durch Einfügen für eine bereits sortierte Datei an. Das Verfahren erkennt für jedes Element sofort, dass es bereits an der richtigen Stelle in der Datei steht, und die Gesamtlaufzeit ist linear. Für Bubblesort gilt das Gleiche; Sortieren durch Auswählen ist nach wie vor quadratisch.

Definition 6.2

Eine Inversion ist ein Paar von Schlüsseln, die in der Datei an falscher Position stehen. ■

Um die Anzahl der Inversionen in einer Datei festzustellen, können wir zum Beispiel für jedes Element die Anzahl der größeren Elemente zählen, die links von ihm stehen (wir bezeichnen diese Größe als die Anzahl der Inversionen, die dem Element entsprechen). Dieser Wert ist aber genau der Abstand, um den die Elemente zu verschieben sind, wenn sie beim Sortieren durch Einfügen in die Datei eingefügt werden. Eine Datei mit einer gewissen Ordnung hat auch weniger Inversionen als eine vollkommen zufällige Datei.

In einem Typ einer teilweise sortierten Datei befindet sich jedes Element schon recht nahe an seiner endgültigen Position in der Datei. Beispielsweise sortieren manche Kartenspieler ihr Blatt, indem sie zuerst die Karten nach der Farbe ordnen und so alle Karten in die Nähe ihrer endgültigen Position bringen, anschließend sortieren sie die Karten einzeln nacheinander. Wir sehen uns eine Reihe von Sortierverfahren an, die fast in der gleichen Weise arbeiten – sie bringen die Elemente in frühen Stufen nahe an ihre endgültige Position, um eine teilweise sortierte Datei zu produzieren, in der jedes Element nicht weit von der Position entfernt ist, wo es letztlich hingehört. Die Verfahren Sortieren durch Einfügen und Bubblesort (nicht jedoch Sortieren durch Auswählen) sind effiziente Methoden für das Sortieren derartiger Dateien.

Eigenschaft 6.4 *Sortieren durch Einfügen und Bubblesort benötigen eine lineare Anzahl von Vergleichen und Vertauschungen für Dateien mit höchstens einer konstanten Anzahl von Inversionen für jedes Element.*

Wie eben erwähnt ist die Laufzeit bei Sortieren durch Einfügen direkt proportional zur Anzahl der Inversionen in der Datei. Bei Bubblesort (hier beziehen wir uns auf Programm

6.14 mit der Modifikation, dass es terminiert, wenn die Datei sortiert ist) ist der Beweis nicht ganz so banal (siehe Übung 6.39). Jeder Bubblesort-Durchgang verringert die Anzahl der kleineren Elemente rechts jedes Elements genau um 1 (sofern die Anzahl nicht bereits 0 war), sodass Bubblesort höchstens eine konstante Anzahl von Durchläufen für die hier betrachteten Arten von Dateien benötigt und demzufolge höchstens eine lineare Anzahl von Vergleichen und Vertauschungen durchführt. ■

Eine teilweise sortierte Datei kann auch dadurch entstehen, dass man an eine sortierte Datei einige Elemente anfügt oder in der sortierten Datei bestimmte Elemente bearbeitet, um ihre Schlüssel zu ändern. Diese Art von Datei ist in Sortieranwendungen weit verbreitet. Hier bietet sich Sortieren durch Einfügen als effizientes Verfahren an, während Bubblesort und Sortieren durch Auswählen nicht geeignet sind.

Eigenschaft 6.5 *Sortieren durch Einfügen benötigt eine lineare Anzahl von Vergleichen und Vertauschungen für Dateien mit höchstens einer konstanten Anzahl von Elementen, die mehr als eine konstante Anzahl von korrespondierenden Inversionen haben.*

Die Laufzeit beim Sortieren durch Einfügen hängt von der Gesamtzahl der Inversionen in der Datei ab, jedoch nicht von der Art und Weise, nach der die Inversionen verteilt sind. ■

Um Schlüsse über die Laufzeit aus den Eigenschaften 6.1 bis 6.5 ziehen zu können, müssen wir die relativen Kosten von Vergleichen und Vertauschungen analysieren, das heißt einen Faktor, der seinerseits von der Größe der Elemente und Schlüssel abhängt (siehe Tabelle 6.1). Wenn zum Beispiel die Elemente aus Schlüsseln zu je einem Wort bestehen, dann sollte ein Austausch (vier Arrayzugriffe) ungefähr doppelt so lange dauern wie ein Vergleich. In einer derartigen Situation unterscheiden sich die Laufzeiten der Verfahren Sortieren durch Einfügen und Sortieren durch Auswählen kaum, während Bubblesort langsamer ist. Wenn jedoch die Elemente groß im Vergleich zu den Schlüsseln sind, schneidet Sortieren durch Auswählen am besten ab.

Eigenschaft 6.6 *Sortieren durch Auswählen benötigt eine lineare Laufzeit für Dateien mit großen Elementen und kleinen Schlüsseln.*

M sei das Verhältnis von Größe des Elements zur Größe des Schlüssels. Dann können wir die Kosten eines Vergleichs als 1 Zeiteinheit und die Kosten einer Vertauschung als M Zeiteinheiten annehmen. Sortieren durch Auswählen benötigt ungefähr $N^2/2$ Zeiteinheiten für Vergleiche und etwa NM Zeiteinheiten für Vertauschungen. Ist M größer als ein konstantes Vielfaches von N, dann ist der Term NM wesentlich größer als der Term N^2, sodass die Laufzeit proportional zu NM und damit proportional zur Zeit für die Bewegung aller Daten ist. ■

Sind zum Beispiel 1000 Elemente, die aus 1-Wort-Schlüsseln und jeweils 1000 Datenwörtern bestehen, zu sortieren und müssen wir die Elemente tatsächlich neu anordnen, haben wir keine bessere Wahl als Sortieren durch Auswählen, da sich die Laufzeit vor allem aus den Kosten ergibt, die für die Verschiebung aller 1 Million Datenwörter erforderlich ist.

Sortieren durch Einfügen und Sortieren durch Auswählen sind ungefähr doppelt so schnell wie Bubblesort für kleine Dateien, aber die Laufzeiten wachsen quadratisch (mit doppelter Dateigröße nimmt die Laufzeit um das Vierfache zu). Keine der Methoden ist für große zufällig geordnete Dateien nützlich – zum Beispiel sind die Werte, die denjenigen in dieser Tabelle entsprechen, kleiner als 2 für den Shellsort-Algorithmus in Abschnitt 6.8. Sind umfangreiche Vergleiche durchzuführen – wenn die Schlüssel zum Beispiel Objekte oder Strings sind –, dann läuft Sortieren durch Einfügen wesentlich schneller als die beiden anderen Verfahren, weil es weniger Vergleiche verwendet. Hier nicht angegeben ist der Fall, bei dem Vertauschungen teuer sind; dann ist nämlich Sortieren durch Auswählen am besten.

	`int`-Elemente				`Integer`-Schlüssel			`String`-Schlüssel		
N	**S**	**I***	**I**	**B**	**S**	**I**	**B**	**S**	**I**	**B**
1000	14	54	8	54	100	55	130	129	65	170
2000	54	218	33	221	436	229	569	563	295	725
4000	212	848	129	871	1757	986	2314	2389	1328	3210

Schlüssel:

S Sortieren durch Auswählen (Programm 6.12)
I* Sortieren durch Einfügen, austauschbasiert (Programm 6.1)
I Sortieren durch Einfügen (Programm 6.13)
B Bubblesort (Programm 6.14)

Tabelle 6.1: Empirische Ergebnisse für elementare Sortierverfahren

ÜBUNGEN

▷ 6.35 Welches der drei elementaren Sortierverfahren (Sortieren durch Auswählen, Sortieren durch Einfügen und Bubblesort) läuft am schnellsten für eine Datei, in der alle Schlüssel identisch sind?

6.36 Welches der drei elementaren Sortierverfahren läuft am schnellsten für eine Datei, die in umgekehrter Reihenfolge sortiert ist?

○ 6.37 Führen Sie empirische Untersuchen für $N = 10$, 100 und 1000 mit Sortieren durch Auswählen, Sortieren durch Einfügen und Bubblesort durch und geben Sie eine Tabelle mit Mittelwert und Standardabweichung für die Anzahl der verwendeten Vergleiche und Vertauschungen aus (siehe Übung 6.12).

6.38 Geben Sie ein Beispiel für eine Datei mit 10 Elementen an (verwenden Sie die Schlüssel `A` bis `J`), für die Bubblesort weniger Vergleiche als Sortieren durch Einfügen benötigt, oder zeigen Sie, dass keine derartige Datei möglich ist.

→

➔ • 6.39 Zeigen Sie, dass jeder Bubblesort-Durchgang die Anzahl der größeren Elemente links jedes Elements genau um 1 verringert (sofern diese Anzahl noch nicht 0 ist).

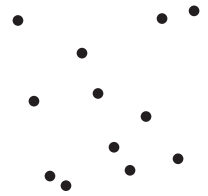
6.40 Implementieren Sie eine Version von Bubblesort, die abwechselnd von links nach rechts und von rechts nach links durch die Daten geht. Dieser (schnellere aber kompliziertere) Algorithmus heißt *Shakersort* (siehe Abbildung 6.8).

• 6.41 Zeigen Sie, dass Eigenschaft 6.5 nicht auf Shakersort (siehe Übung 6.40) zutrifft.

6.7 Visualisierung von Algorithmen

Sortieralgorithmen eignen sich besonders für bildliche Darstellungen, um die dynamischen Abläufe deutlich zu machen. In den Abbildungen dieses Kapitels haben Sie bereits mehrere Beispiele gesehen, die diese Tatsache unterstreichen. Visuelle Darstellungen sind einerseits sehr aufschlussreich, um die Sortieralgorithmen zu verstehen, und andererseits leicht zu erzeugen, sodass wir hier kurz abschweifen, um uns diesem Thema zu widmen.

Vielleicht am einfachsten lassen sich Visualisierungen von Sortieralgorithmen mit PostScript (siehe Abschnitt 4.3) erzeugen, wie es Abbildung 6.9 zeigt. Kaum etwas ist unkomplizierter, als ein Array von Punkten anhand ihrer Koordinaten grafisch darzustellen. Ähnlich einfach ist es, ein Java-Programm zu implementieren, das derartige PostScript-Programme erzeugt. Man muss lediglich den Inhalt eines Arrays periodisch ausgeben, wie es Programm 6.15 demonstriert.



```
/w 72 def /h 72 def N 12 def
/dot
 {
   moveto currentpoint
   2 0 360 arc fill
 } def
 0 .91 dot
 1 .49 dot
 2 .10 dot
 3 .05 dot
 4 .74 dot
 5 .53 dot
 6 .25 dot
 7 .13 dot
 8 .41 dot
 9 .92 dot
10 .19 dot
11 .96 dot
```

Abbildung 6.9: PostScript-Visualisierung eines Array-Inhalts

Dieses PostScript-Programm zeichnet den Inhalt eines Arrays von Zahlen zwischen 0 und 1 durch Aufruf der Funktion `dot` *mit dem Index und dem Wert jedes Arrayeintrags als Argumente. Die Werte* `w` *und* `h` *geben die Breite und Höhe des Rahmens an;* `N` *ist die Anzahl der Werte im Array. Die Funktion* `dot` *skaliert den Index auf eine x-Koordinate zwischen* `0` *und* `w`*, indem sie mit* `w` *multipliziert und durch* `N` *dividiert, und den Wert in eine y-Koordinate zwischen* `0` *und* `h`*, indem sie mit* `h` *multipliziert. Dann zeichnet das Programm einen gefüllten Kreis vom Radius 2 um den Punkt (x, y).*

Programm 6.15 Visualisierung eines Sortieralgorithmus

Dieses Programm gibt ein PostScript-Programm wie das in Abbildung 6.9 aus. Die Konstanten `h` und `w` spezifizieren die Höhe und Breite des Rahmens (in der Maßeinheit Punkt). Die Konstante `cuts` legt die Anzahl der Rahmen fest. Die Methode `show` gibt den Code aus, der die Werte im Array zeichnet. Die Methode `sort` ruft `show` mit geeigneten Intervallen auf.

```
class Visualize
  { private static final int cuts = 5, h = 72, w = 72;
    static int N;
    static void sort(double a[], int l, int r)
      { double t;
        for (int i = l; i <= r; i++)
          { for (int j = i; j > l; j--)
              if (a[j] < a[j-1]) exch(a, j-1, j);
            if (i*cuts % N == 0) show(a, l, r);
          }
      }
    static void show(double[] a, int l, int r)
      {
        for (int i = l; i <= r; i++)
          { float x = h*((float) i) / ((float) N);
            float y = w*((float) a[i]);
            Out.println(x + " " + y + " dot");
          }
        Out.println(1.1*w + " 0 translate");
      }
    public static void main(String[] args)
      { N = Integer.parseInt(args[0]);
        double a[] = new double[N];
        for (int i = 0; i < N; i++)
          a[i] = Math.random();
        Out.print("72 72 translate ");
        Out.print("/dot {moveto currentpoint");
        Out.println(" 2 0 360 arc fill} def");
        sort(a, 0, N-1);
      }
  }
```

Natürlich bietet Java auch ein `Graphics`-Objekt mit entsprechenden Zeichenmethoden und wir könnten diese direkt einsetzen, um eine Visualisierung zu erzeugen (siehe Übung 6.42). Allerdings überlassen wir das dem Leser als Übung, weil es in Java interessanter und nicht wesentlich schwieriger ist, *dynamische* visuelle Darstellungen zu betrachten, bei denen wir während des Sortiervorgangs beobachten können, wie die Elemente durch das Array laufen.

Die Programme 6.16 und 6.17 implementieren als Beispiel eine Java-Sortieranimation in einem Java-Applet. Die Implementierung basiert darauf, die Methode `exch` so zu erweitern, dass sie die visuelle Darstellung jedes Mal ändert, wenn der Sortieralgorithmus eine Vertauschung vornimmt.

Programm 6.16 Animation eines Sortieralgorithmus

Diese abstrakte Klasse stellt die grundlegenden Methoden bereit, um die Animation von Sortieralgorithmen zu unterstützen (siehe Programm 6.17).

```
import java.applet.Applet;
import java.awt.*;
abstract public class Animate
  extends Applet implements Runnable
  { Graphics g;
    Thread animatorThread;
    int N; double[] a;
    public void start()
      { g = getGraphics();
        new Thread(this).start();
      }
    public void stop() { animatorThread = null; }
    public void run()
      { N = Integer.parseInt(getParameter("N"));
        a = new double[N];
        for (int i = 0; i < N; i++)
          { a[i] = Math.random();
            dot(X(i), Y(a[i]), Color.black); }
        sort(a, 0, N-1);
      }
    int X(int i)
      { return (i*getSize().width)/N; }
    int Y(double v)
      { return (1.0 - v)*getSize().height; }
    void dot(int x, int y, Color c)
      { g.setColor(c); g.fillOval(x, y, 5, 5); }
    void exch(double [] a, int i, int j)
      { double t = a[i]; a[i] = a[j]; a[j] = t;
        dot(X(i), Y(a[j]), Color.red);
        dot(X(j), Y(a[i]), Color.red);
        dot(X(i), Y(a[i]), Color.black);
        dot(X(j), Y(a[j]), Color.black);
      }
    abstract void sort(double a[], int l, int r);
  }
```

Programm 6.17 Animation eines Sortieralgorithmus

Diese Klasse, die Programm 6.16 erweitert, animiert das Sortieren durch Einfügen von Programm 6.6 und liefert eine visuelle Darstellung auf dem Bildschirm analog zu den Abbildungen 6.6 und 6.8, verschiebt aber die Punkte an ihren Platz. Die Punkte hinterlassen außerdem Spuren, sodass die resultierende Anzeige einen Verlauf des Sortiervorgangs liefert. Da die Implementierung auf einer Erweiterung der Methode `exch` basiert, wirkt sie unmittelbar für jedes austauschbasierte Sortierprogramm. Es empfiehlt sich, auch die Methode `less` entsprechend zu erweitern (siehe Übung 6.44).

```
import java.awt.*;
public class SortAnimate extends Animate
  {
    void sort(double a[], int l, int r)
      {
        for (int i = l+1; i <= r; i++)
          for (int j = i; j > l; j--)
            if (a[j] < a[j-1]) exch(a, j-1, j);
      }
  }
```

Auch wenn einige Details anders aussehen, können Sie die Animation normalerweise im Appletviewer oder in einem Browser mit einer .html-Datei ausführen, die den folgenden HTML-Code enthält:

```
<applet code=SortAnimate.class width=640 height=480>
<param name=N value="200">
</applet>
```

In vielen Java-Implementierungen bietet diese einfache Strategie eine recht wirkungsvolle Animation; bei anderen läuft die Animation vielleicht nicht ganz reibungslos, was von der Systembelastung, den Pufferstrategien und anderen Faktoren abhängt.

Es geht über den Rahmen dieses Buchs hinaus, alle Fragen in Bezug auf die Entwicklung effektiver Animationen zu untersuchen; andererseits ist es den Aufwand wert, eine grundlegende Animation wie diese zum Laufen zu bringen. Beispielsweise erhalten wir mit Programm 6.17 eine ansprechende Animation für jeden austauschbasierten Sortieralgorithmus, ohne den Sortiercode in irgendeiner Form ändern zu müssen.

Beachten Sie, dass wir aus Visualisierungen und Animationen keine Rückschlüsse auf die Leistung ziehen können, sofern wir sie nicht sorgfältig erstellt haben. Wenn wir zum Beispiel Sortieren durch Auswählen mit Programm 6.16 animieren, ist die Animation weitaus schneller als für Sortieren durch Einfügen abgeschlossen. Wie bereits erwähnt hängt das einfach damit zusammen, dass Sortieren durch Auswählen nur wenige Austauschoperationen ausführt. Wenn wir sowohl Vergleiche als auch Vertauschungen in die Animation einbeziehen, liefern die Animationen ein repräsentativeres Bild von der Leistung der Algorithmen.

Oftmals beobachten wir sogar eher die Leistungseigenschaften eines Graphiksystems als die unserer Algorithmen. Beispielsweise lässt sich die Animation in den Programmen 6.16 und 6.17 für kleine Beispiele flüssiger darstellen, wenn man die `repaint()`-Operation (Neuzeichnen) regelmäßig aufruft. Allerdings kann `repaint()` übermäßig teuer sein: wollen wir beispielsweise einen Algorithmus animieren, der Millionen von Vertauschungen durchführt, lässt das Neuzeichnen bei jeder Vertauschung aufgrund der erforderlichen Zeit für Millionen (oder selbst Tausende) von `repaint()`-Operationen das System zusammenbrechen.

Abgesehen von solchen Fallstricken kann die visuelle Darstellung eine wichtige Rolle spielen, um die Funktionsweise von Algorithmen besser zu verstehen, und wir setzen sie großzügig für die Abbildungen im gesamten Buch ein. Wir hätten auch für jeden Algorithmus Übungen angeben können, die es Ihnen zur Aufgabe machen, die Algorithmen zu visualisieren und zu animieren, weil sich jeder Algorithmus, jede visuelle Darstellung und jede Animation in allen Sortierverfahren auf interessante Weise verbessern ließe. Wir haben aber davon abgesehen, weil die meisten Leser ohnehin mit visuellen Darstellungen arbeiten werden, wenn es angebracht ist, was insbesondere für Sortieralgorithmen zutrifft. Nachdem Sie einmal ein Programm wie Programm 6.15 oder die Programme 6.16 und 6.17 eingesetzt haben, finden Sie sicherlich Gefallen daran, auch andere Algorithmen zu visualisieren.

ÜBUNGEN

6.42 Schreiben Sie eine Version von Programm 6.15, die geeignete Methoden der Java-Klasse `Graphics2D` verwendet, um die Visualisierung auf Ihrem Bildschirm auszugeben.

• 6.43 Implementieren Sie Sortieren durch Einfügen, Sortieren durch Auswählen und Bubblesort in PostScript und verwenden Sie Ihre Implementierung, um Abbildungen analog zu den Abbildungen 6.6 bis 6.8 zu zeichnen. Probieren Sie entweder eine rekursive Implementierung aus oder konsultieren Sie das Handbuch, um sich mit Schleifen und Arrays in PostScript vertraut zu machen.

○ 6.44 Modifizieren Sie die Methode `sort` in Programm 6.17, um Schlüssel mit `less` zu vergleichen. Erweitern Sie Programm 6.16 durch eine Implementierung von `less`, damit das Programm eine dynamische visuelle Darstellung von Vergleichs- und Austauschoperationen ausgibt. Animieren Sie sowohl Sortieren durch Einfügen als auch Sortieren durch Auswählen.

• 6.45 Entwickeln Sie ein interaktives Animationsprogramm, bei dem der Benutzer den zu animierenden Algorithmus aus einer Liste auswählen, die Anzahl der zu sortierenden Elemente spezifizieren und das Modell für das Erzeugen zufälliger Elemente (siehe Übung 6.17) festlegen kann.

→

➔ •• 6.46 Erweitern Sie Ihr Programm von Übung 6.45, um ein Skript aller animierten Ereignisse zu speichern. Implementieren Sie dann die Funktionalität, damit der Benutzer Algorithmen wiederholen und sie rückwärts laufen lassen kann.

6.8 Shellsort

Sortieren durch Einfügen ist langsam, weil dieser Algorithmus nur die benachbarten Elemente austauscht, sodass sich die Elemente bei jedem Durchgang nur jeweils um eine Position durch das Array bewegen können. Wenn zum Beispiel das Element mit dem kleinsten Schlüssel am Ende des Arrays steht, sind N Schritte erforderlich, um es an die Position zu bringen, wo es hingehört. Eine einfache Erweiterung zum Sortieren durch Einfügen ist der Algorithmus *Shellsort*, der Vertauschungen von weiter auseinander liegenden Elementen erlaubt und dadurch eine höhere Geschwindigkeit erzielt.

Dem Algorithmus liegt die Idee zugrunde, die Datei so umzuordnen, dass man eine sortierte Datei erhält, wenn man jedes h-te Element entnimmt (wobei man bei einem beliebigen Element beginnt). Eine derartige Datei heißt *h-sortiert*. Anders ausgedrückt besteht eine h-sortierte Datei aus h unabhängig sortierten Dateien, die ineinander verschachtelt sind. Durch h-Sortieren für große Werte von h können wir Elemente im Array über größere Entfernungen hinweg verschieben und somit das h-Sortieren für kleinere Werte von h erleichtern. Wendet man eine derartige Prozedur auf eine beliebige Folge von h-Werten an, die mit 1 endet, erhält man eine sortierte Datei: Das sind die Grundzüge von Shellsort.

Um Shellsort zu implementieren, kann man für jedes h das Sortieren durch Einfügen unabhängig auf jede der h Teildateien anwenden. Trotz der scheinbaren Einfachheit dieses Verfahrens können wir einen sogar noch einfacheren Ansatz verwenden, und zwar genau deshalb, weil die Teildateien unabhängig sind. Wenn wir die Datei h-sortieren, fügen wir sie einfach unter die vorherigen Elemente in ihrer h-Teildatei ein, indem wir größere Elemente nach rechts verschieben

A S O R T I N G E X A M P L E
A S O R E I N G T X A M P L E
A S O R E I N G P X A M T L E

A I O R E S N G P X A M T L E
A I O R E S N G P X A M T L E
A I O R E L N G P S A M T X E

A I N R E L O G P S A M T X E
A I A R E L N G P S O M T X E
A I A R E L E G P S N M T X O

A I A G E L E R P S N M T X O
A I A G E L E M P S N R T X O

A S O R T I N G E X A M P L E
A I O R T S N G E X A M P L E
A I N R T S O G E X A M P L E
A I N G T S O R E X A M P L E
A I N G E S O R T X A M P L E
A I N G E S O R T X A M P L E
A I A G E S N R T X O M P L E
A I A G E S N M T X O R P L E
A I A G E S N M P X O R T L E
A I A G E L N M P S O R T X E
A I A G E L E M P S N R T X O

Abbildung 6.10: Sortieren mit einer Sprungweite von 4

Der obere Teil der Abbildung zeigt den Ablauf beim 4-Sortieren einer Datei von 15 Elementen: Zuerst wird die Teildatei an den Positionen 0, 4, 8, 12 sortiert, danach die Teildatei an den Positionen 1, 5, 9, 13 , dann die Teildatei an den Positionen 2, 6, 10, 14 und schließlich die Teildatei an den Positionen 3, 7, 11 (jeweils mit Sortieren durch Einfügen). Da die vier Teildateien unabhängig sind, können wir das gleiche Ergebnis erreichen, indem wir jedes Element an seine Position in seiner Teildatei bringen und jeweils um vier Elemente zurückgehen (unten). Entnimmt man aus der oberen Darstellung die erste Zeile aus jedem Abschnitt, dann die zweite Zeile aus jedem Abschnitt usw., gelangt man zur unteren Darstellung.

(siehe Abbildung 6.10). Diese Aufgabe realisieren wir, indem wir den Code für das Sortieren durch Einfügen verwenden, aber so modifizieren, dass wir um *h* anstelle von 1 inkrementieren oder dekrementieren, wenn wir durch die Datei gehen. Diese Beobachtung reduziert die Shellsort-Implementierung auf nichts weiter als einen Durchgang à la Sortieren durch Einfügen durch die Datei für jedes Inkrement, wie es Programm 6.18 zeigt. Abbildung 6.11 veranschaulicht die Arbeitsweise dieses Programms.

Programm 6.18 Shellsort

Wenn wir keine Markierungsschlüssel verwenden und dann beim Sortieren durch Einfügen für jedes Auftreten von »1« ein »h« setzen, führt das resultierende Programm ein *h*-Sortieren der Datei durch. Fügt man noch eine äußere Schleife hinzu, um die Abstände zu ändern, erhält man diese kompakte Shellsort-Implementierung, die folgende Abstandsfolge verwendet: 1 4 13 40 121 364 1093 3280 9841 …

```
static void shell(ITEM[] a, int l, int r)
  { int h;
    for (h = 1; h <= (r-l)/9; h = 3*h+1);
    for ( ; h > 0; h /= 3)
      for (int i = l+h; i <= r; i++)
        { int j = i; ITEM v = a[i];
          while (j >= l+h && less(v, a[j-h]))
            { a[j] = a[j-h]; j -= h; }
          a[j] = v;
        }
  }
```

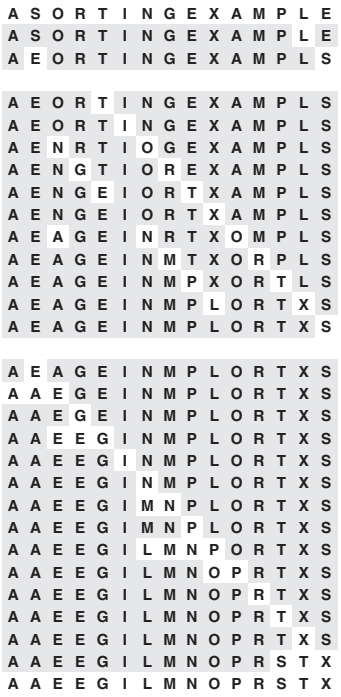


Abbildung 6.11:
Beispiel für Shellsort

Sortiert man eine Datei mit 13-Sortieren (oben), dann 4-Sortieren (Mitte), schließlich 1-Sortieren (unten), sind nicht viele Vergleiche notwendig (wie es die nicht schattierten Elemente anzeigen). Der letzte Durchgang ist lediglich ein Sortieren durch Einfügen, wobei aber kein Element weit zu verschieben ist, weil die beiden ersten Durchläufe bereits eine gewisse Ordnung in die Datei gebracht haben.

Wie entscheiden wir nun, welche Abstandsfolge zu verwenden ist? Im Allgemeinen ist diese Frage schwer zu beantworten. In der Literatur sind viele unterschiedliche Abstandsfolgen untersucht worden, wobei sich einige in der Praxis bewährt haben. Eine nachweisbar optimale Folge ist jedoch nicht gefunden worden. In der Praxis verwenden wir im Allgemeinen Folgen, die etwa geometrisch abnehmen, sodass die Anzahl der Abstandswerte logarithmisch zur Größe der Datei ist. Wenn zum Beispiel jeder Abstandswert ungefähr die Hälfte des vorherigen ist, dann benöti-

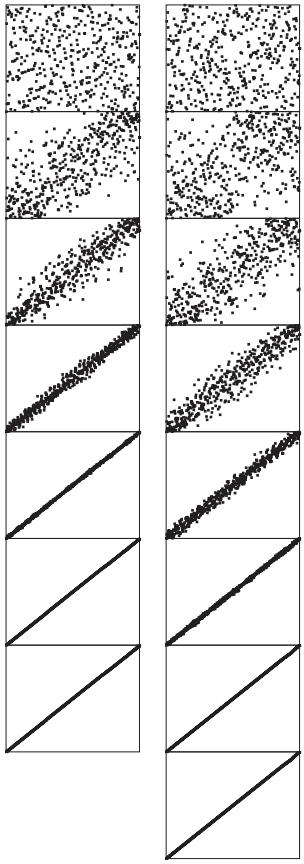


Abbildung 6.12: Dynamische Eigenschaften von Shellsort (zwei unterschiedliche Abstandsfolgen)

In dieser Darstellung eines laufenden Shellsort-Algorithmus kann man sich ein schwingendes Gummiband vorstellen, das an den Ecken eingespannt ist und die Punkte in Richtung der Diagonalen stößt. Die Bilder zeigen zwei Abstandsfolgen: 121 40 13 4 1 (links) und 209 109 41 19 5 1 (rechts). Die zweite erfordert einen Durchgang mehr als die erste, läuft aber schneller, weil jeder Durchgang effizienter ist.

gen wir nur etwa 20 Abstände, um eine Datei mit 1 Million Elementen zu sortieren; ist das Verhältnis etwa 1 zu 4, genügen 10 Abstände. Leicht einzusehen ist die wichtige Maßgabe, möglichst wenig Abstände zu verwenden – wir müssen auch arithmetische Wechselwirkungen zwischen den Abständen berücksichtigen, beispielsweise die Größe ihrer gemeinsamen Teiler und andere Eigenschaften.

Eine gute Abstandsfolge bringt in der Praxis höchstens eine 25-prozentige Geschwindigkeitssteigerung, das Problem ist aber insofern interessant, weil es zeigt, dass ein offensichtlich einfacher Algorithmus auch sehr komplex sein kann.

Die in Programm 6.18 verwendete Sequenz 1 4 13 40 121 364 1093 3280 9841 … mit einem Verhältnis der Abstände von etwa 1 zu 3 wurde 1969 von Donald E. Knuth vorgeschlagen (siehe den Referenzabschnitt). Diese Folge lässt sich leicht berechnen (man beginnt mit 1 und generiert den nächsten Abstand durch Multiplikation mit 3 und Addieren von 1) und führt zu einem relativ effizienten Sortieren, selbst für mittelgroße Dateien, wie es die Abbildungen 6.12 und 6.13 verdeutlichen.

Viele andere Abstandsfolgen führen zu einem effizienteren Sortieren, doch ist es selbst für relativ große N schwierig, die Sequenz in Programm 6.18 um mehr als 20 Prozent zu übertreffen. Dazu gehört die Abstandsfolge 1 8 23 77 281 1073 4193 16577…, die sich aus der Vorschrift $4^{i+1} + 3*2^i + 1$ für $i > 0$ berechnet und nachweisbar ein schnelleres Verhalten im ungünstigsten Fall zeigt (siehe Eigenschaft 6.10). Abbildung 6.12 zeigt, dass diese Sequenz und die von Knuth vorgeschlagene Sequenz – sowie viele andere Sequenzen – bei großen Dateien ähnliche dynamische Eigenschaften haben. Die Möglichkeit, dass noch bessere Abstandsfolgen existieren, ist nach wie vor gegeben. Die Übungen geben einige Anregungen zu verbesserten Abstandsfolgen.

Andererseits gibt es einige schlechte Abstandsfolgen: Zum Beispiel führt 1 2 4 8 16 32 64 128 256 512 1024 2048 … (die ursprünglich von Shell vorgeschlagene Sequenz, als er 1959 seinen Algorithmus vorstellte (siehe den Referenzabschnitt)) höchstwahrscheinlich zu einer schlechten Leistung, weil Elemente an unge-

raden Positionen bis zum letzten Durchgang nicht mit Elementen an geraden Positionen verglichen werden. Bei zufälligen Dateien ist dieser Effekt spürbar, im ungünstigsten Fall ist er katastrophal: Das Verfahren entartet und benötigt eine quadratische Laufzeit, wenn beispielsweise die Hälfte der Elemente mit den kleinsten Werten an geraden Positionen steht und die Hälfte der Elemente mit den größten Werten an ungeraden Positionen (siehe Übung 6.50).

Programm 6.18 berechnet den nächsten Abstand, indem es den aktuellen Wert durch 3 dividiert; mit einem festgelegten Anfangswert stellt es sicher, dass immer die gleiche Folge verwendet wird. Alternativ kann man den Anfangswert einfach mit `h = N/3` oder mit einer anderen Funktion von N bestimmen. Solche Strategien sollte man jedoch vermeiden, da sie bei bestimmten Werten von N wahrscheinlich zu einer der schlechten Folgen führen können, wie sie der vorherige Absatz beschrieben hat.

Die Effizienz von Shellsort können wir zwangsläufig nur ungenau beschreiben, weil noch niemand den Algorithmus analysieren konnte. Diese Wissenslücke erschwert es nicht nur, unterschiedliche Abstandsfolgen auszuwerten, sondern auch Shellsort mit anderen Sortierverfahren analytisch zu vergleichen. Nicht einmal die funktionelle Abhängigkeit der Laufzeit ist für Shellsort bekannt (außerdem ist ihre Gestalt von der Abstandsfolge abhängig). Knuth hat als funktionelle Formen $N(\log N)^2$ und $N^{1{,}25}$ gefunden, die sich beide gut an die Daten anpassen, und spätere Untersuchungen geben die These an, dass eine kompliziertere Funktion der Form $N^{1+1/\sqrt{\lg N}}$ bei bestimmten Sequenzen beteiligt ist.

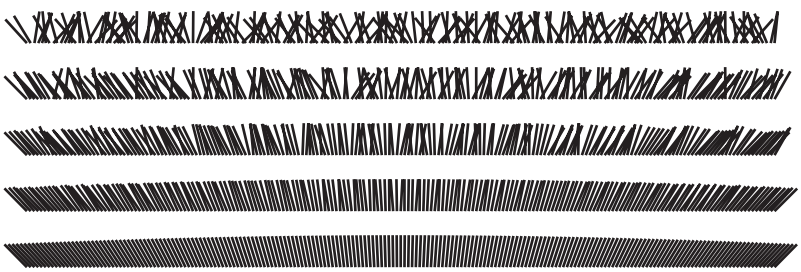


Abbildung 6.13: Eine zufällige Permutation mit Shellsort sortieren

Die Wirkung jedes Durchgangs in Shellsort besteht darin, die Datei als Ganzes näher an die sortierte Reihenfolge zu bringen. Die Datei wird 40-sortiert, dann 13-sortiert, danach 4-sortiert und schließlich 1-sortiert. Jeder Durchgang bringt die Datei näher an die sortierte Reihenfolge.

Am Ende dieses Abschnitts schweifen wir noch etwas ab, um verschiedene Fakten über die Analyse von Shellsort zu behandeln, die *bekannt* sind. Wir wollen damit vor allem verdeutlichen, dass sogar offensichtlich einfache Algorithmen komplexe Eigenschaften haben können und dass die Analyse von Algorithmen nicht nur von praktischer Bedeutung ist, sondern auch intellektuell herausfordernd sein kann. Leser, die an der Suche nach einer neuen und verbesserten Abstandsfolge für Shellsort interessiert sind, finden die folgenden Informationen sicherlich nützlich; andere Leser können ohne weiteres zu Abschnitt 6.9 weitergehen.

Eigenschaft 6.7 *Das Ergebnis des h-Sortierens einer k-sortierten Datei ist eine Datei, die sowohl h- als auch k-sortiert ist.*

Diese Aussage scheint klar zu sein, lässt sich aber nur schwer beweisen (siehe Übung 6.61). ■

Eigenschaft 6.8 *Shellsort benötigt weniger als $N(h-1)(k-1)/g$ Vergleiche, um eine Datei, die h- und k-sortiert ist, mit g-Sortieren zu sortieren, vorausgesetzt dass h und k teilerfremd sind.*

Die Grundlage für diese Aussage ist in Abbildung 6.14 veranschaulicht. Kein Element weiter als $(h-1)(k-1)$ Positionen links von einem gegebenen Element x kann größer als x sein, wenn h und k teilerfremd sind (siehe Übung 6.57). Beim g-Sortieren untersuchen wir höchstens eines je g dieser Elemente. ■

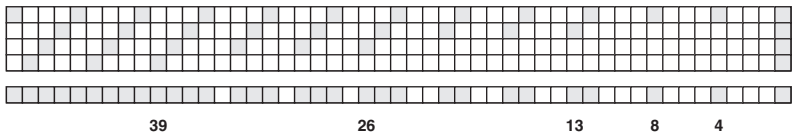


Abbildung 6.14: Eine 4-und 13-sortierte Datei

Die untere Zeile zeigt ein Array, indem die grauen Elemente kleiner oder gleich dem Element am rechten Rand sein müssen, wenn das Array sowohl 4- als auch 13-sortiert ist. Die vier oberen Zeilen geben das ursprüngliche Muster an. Wenn sich das rechte Element an der Arrayposition i befindet, bedeutet 4-Sortieren, dass Elemente an Arraypositionen $i-4$, $i-8$, $i-12$, … kleiner oder gleich sind (oben); 13-Sortieren bedeutet, dass das Element an $i-13$ und aufgrund des 4-Sortierens die Elemente an $i-17$, $i-21$, $i-25$, … kleiner oder gleich sind (zweite Zeile von oben); außerdem sind das Element an $i-26$ und aufgrund des 4-Sortierens die Elemente an $i-30$, $i-34$, $i-38$, … kleiner oder gleich (dritte Zeile von oben) usw. Die verbleibenden weißen Quadrate sind diejenigen, die größer sein können als das linke Element; es gibt höchstens 18 solche Elemente (und das am weitesten entfernte Element an $i-36$). Somit sind höchstens $18N$ Vergleiche für ein Sortieren durch Einfügen einer 13-sortierten und 4-sortierten Datei der Größe N notwendig.

Eigenschaft 6.9 *Shellsort benötigt für die Abstände 1 4 13 40 121 364 1093 3280 9841 … weniger als $O(N^{3/2})$ Vergleiche.*

Bei großen Abständen gibt es h Teildateien einer Größe von ungefähr N/h bei Kosten im ungünstigsten Fall von etwa N^2/h. Bei kleinen Abständen besagt Eigenschaft 6.8, dass die Kosten ungefähr Nh betragen. Das Ergebnis folgt, wenn wir die bessere dieser Schranken für jeden Abstand verwenden. Es gilt für jede teilerfremde Folge, die exponentiell wächst. ■

Eigenschaft 6.10 *Shellsort benötigt für die Abstände 1 8 23 77 281 1073 4193 16577 … weniger als $O(N^{4/3})$ Vergleiche.*

Der Beweis dieser Eigenschaft ergibt sich aus dem Beweis von Eigenschaft 6.9. Die Eigenschaft analog zu Eigenschaft 6.8 besagt, dass die Kosten für kleine Abstände ungefähr $Nh^{1/2}$

betragen. Ein Beweis dieser Eigenschaft verlangt nach der Zahlentheorie, was aber über den Rahmen dieses Buchs hinausgeht (siehe den Referenzabschnitt). ■

Die bislang behandelten Abstandsfolgen sind effizient, weil aufeinander folgende Elemente teilerfremd sind. Eine andere Familie von Abstandsfolgen ist genau deshalb effizient, weil aufeinander folgende Elemente *nicht* teilerfremd sind.

Insbesondere besagt der Beweis von Eigenschaft 6.8, dass sich jedes Element in einer 2-sortierten und 3-sortierten Datei während des abschließenden Sortierens durch Einfügen höchstens um eine Position bewegt. Das heißt, eine derartige Datei lässt sich in einem Durchgang von Bubblesort sortieren (die zusätzliche Schleife beim Sortieren durch Einfügen ist nicht notwendig). Wenn nun eine Datei 4-sortiert und 6-sortiert ist, folgt auch, dass sich jedes Element höchstens um eine Position bewegt, wenn wir sie 2-sortieren (weil jede Teildatei 2-sortiert und 3-sortiert ist); und wenn eine Datei 6-sortiert und 9-sortiert ist, bewegt sich jedes Element um höchstens eine Position, wenn wir sie 3-sortieren. Setzt man diese Kette der Schlussfolgerungen fort, kommen wir zu folgendem Dreieck mit Abstandswerten:

```
                1
              2   3
            4   6   9
          8  12  18  27
       16  24  36  54  81
    32  48  72 108 162 243
64  96 144 216 324 486 729
```

Jede Zahl im Dreieck stellt das Doppelte der Zahl rechts darüber und auch das Dreifache der Zahl links darüber dar. Wenn wir diese Zahlen von unten nach oben und von rechts nach links als Abstandsfolge für Shellsort verwenden, dann gehen jedem Abstand x nach der unteren Zeile ein $2x$ und $3x$ voraus, sodass jede Teildatei 2-sortiert und 3-sortiert wird und sich keine Elemente um mehr als eine Position während des gesamten Sortierens bewegen!

Eigenschaft 6.11 *Shellsort benötigt für die Abstände 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81 … weniger als $O(N(\log N)^2)$ Vergleiche.*

Die Anzahl der Abstände im Dreieck, die kleiner als N sind, ist zweifellos kleiner als $(\log_2 N)^2$. ■

Diese Menge von Abstandswerten hat Pratt 1971 entwickelt (siehe den Referenzabschnitt). Es zeigt sich, dass die Abstände von Pratt in der Praxis nicht so gut abschneiden wie die anderen, weil es zu viele davon gibt. Wir können aber nach dem gleichen Prinzip eine Abstandsfolge aus *beliebigen* zwei teilerfremden Zahlen h und k aufbauen. Derartige Sequenzen funktionieren in der Praxis gut, weil die Schranken entsprechend Eigenschaft 6.11 die Kosten für zufällige Dateien überbewerten.

Das Problem, gute Abstandsfolgen für Shellsort zu entwerfen, liefert ein gutes Beispiel für das komplexe Verhalten eines einfachen Algorithmus. Sicherlich können wir uns in dieser Detailtiefe nicht mit allen Algorithmen beschäftigen (einerseits reicht der Platz nicht aus, anderseits stoßen wir an Grenzen, die mathematische Analysen verlangen, die über das Ziel dieses Buchs hinausgehen oder selbst wieder Anstoß für neue Probleme der Forschung bieten). Allerdings sind viele Algorithmen in diesem Buch das Ergebnis umfassender analytischer und empirischer Studien, die viele Informatiker über die letzten Jahrzehnte durchgeführt haben, und wir können von dieser Arbeit profitieren. Diese Untersuchungen veranschaulichen, dass die Suche nach verbesserter Leistung sowohl intellektuell anspruchsvoll als auch praktisch lohnend ist, was selbst für einfache Algorithmen gilt. Die empirischen Ergebnisse in Tabelle 6.2 sollen zeigen, dass verschiedene Ansätze geeignet sind, in der Praxis gut funktionierende Abstandsfolgen zu entwerfen; die relativ kurze Folge 1 8 23 77 281 1073 4193 16577 … gehört zu den einfachsten, die man in einer Shellsort-Implementierung verwendet.

Shellsort ist viele Male schneller als andere elementare Verfahren, selbst wenn die Abstände Potenzen von 2 sind. Bestimmte Abstandsfolgen können den Algorithmus jedoch um einen zusätzlichen Faktor von 5 oder mehr beschleunigen. Die drei besten Folgen in dieser Tabelle unterscheiden sich grundsätzlich im Aufbau. Shellsort ist ein praktisches Verfahren selbst für große Dateien, besonders im Gegensatz zum Sortieren durch Auswählen, Sortieren durch Einfügen und Bubblesort (siehe Tabelle 6.1).

N	O	K	G	S	P	I
12500	68	28	29	29	33	28
25000	117	40	42	38	53	40
50000	280	92	97	75	114	82
100000	1039	217	185	171	252	186
200000	2853	488	468	388	592	418

Schlüssel:

O 1 2 4 8 16 32 64 128 256 512 1024 2048 . . .
K 1 4 13 40 121 364 1093 3280 9841 . . . (Eigenschaft 6.9)
G 1 2 4 10 23 51 113 249 548 1207 2655 5843 . . . (Übung 6.54)
S 1 8 23 77 281 1073 4193 16577 . . . (Eigenschaft 6.10)
P 1 7 8 49 56 64 343 392 448 512 2401 2744 . . . (Übung 6.58)
I 1 5 19 41 109 209 505 929 2161 3905 . . . (Übung 6.59)

Tabelle 6.2: Empirische Ergebnisse für Abstandsfolgen für Shellsort

Abbildung 6.15 zeigt, dass Shellsort bei verschiedenartigsten Dateien akzeptable Ergebnisse liefert, nicht nur für zufällige Dateien. In der Tat ist es eine Herausforderung, eine Datei zu konstruieren, bei der Shellsort für eine gegebene Abstandsfolge langsam läuft (siehe Übung 6.56). Wie bereits erwähnt gibt es einige schlechte Abstandsfolgen, für die

Shellsort im ungünstigsten Fall eine quadratische Anzahl von Vergleichen benötigt (siehe Übung 6.50), doch wurden für eine breite Palette von Sequenzen wesentlich niedrigere Schranken angegeben.

Shellsort ist die bevorzugte Methode für viele Sortieranwendungen, weil dieser Algorithmus selbst für relativ große Dateien akzeptable Laufzeiten aufweist und nur wenig Code benötigt, der sich leicht zum Laufen bringen lässt. Die nächsten Kapitel stellen effizientere Verfahren vor, die aber (außer für große *N*) bestenfalls doppelt so schnell, jedoch wesentlich komplizierter sind. Kurz gesagt, wenn Sie ein Sortierproblem schnell lösen und sich nicht mit der Schnittstelle zu einem Systemsortierverfahren herumschlagen wollen, *verwenden Sie Shellsort* und entscheiden Sie später selbst, ob sich der Aufwand lohnen würde, dieses Verfahren durch ein komplizierteres zu ersetzen.

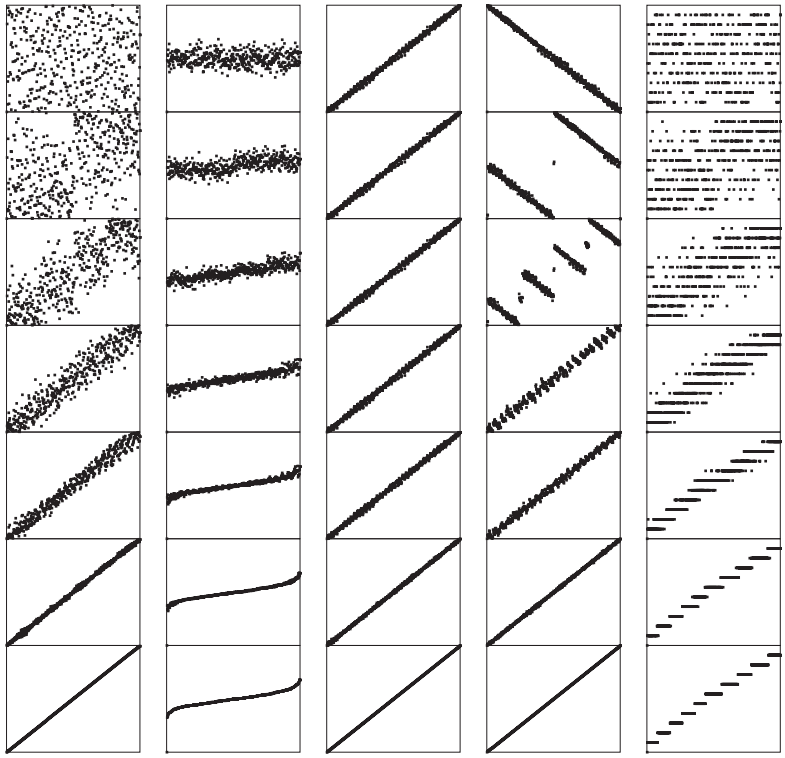


Abbildung 6.15: Dynamische Eigenschaften von Shellsort für verschiedene Arten von Dateien

Diese Diagramme zeigen Shellsort mit den Abständen 209 109 41 19 5 1 für Dateien, die zufällig, Gaußverteilt, nahezu sortiert, nahezu umgekehrt sortiert und zufällig geordnet mit 10 verschiedenen Schlüsselwerten sind (von links nach rechts, in der oberen Reihe). Die Laufzeit für jeden Durchgang hängt davon ab, wie gut die Datei am Anfang dieses Durchgangs sortiert ist. Nach wenigen Durchläufen sind diese Dateien ähnlich sortiert; folglich ist die Laufzeit nicht sonderlich von den Eingabedaten abhängig.

ÜBUNGEN

▷ 6.47 Ist Shellsort stabil?

6.48 Zeigen Sie, wie ein Shellsort mit den Abständen `1 8 23 77 281 1073 4193 16577 ...` zu implementieren ist, wobei die aufeinander folgenden Abstände ähnlich dem Code für die Abstände nach Knuth berechnet werden sollen.

▷ 6.49 Geben Sie Diagramme entsprechend den Abbildungen 6.10 und 6.11 für die Schlüssel `E A S Y Q U E S T I O N` an.

6.50 Ermitteln Sie die Laufzeit, wenn Sie Shellsort mit den Abständen `1 2 4 8 16 32 64 128 256 512 1024 2048 ...` verwenden, um eine Datei mit den Ganzzahlen 1, 2, …, N an den ungeraden Positionen und $N + 1$, $N + 2$, …, $2N$ an den geraden Positionen zu sortieren.

6.51 Schreiben Sie ein Testprogramm, um Abstandsfolgen für Shellsort zu vergleichen. Lesen Sie die Folgen von der Standardeingabe (je Zeile eine Folge) und sortieren Sie dann mit diesen Folgen 10 zufällige Dateien der Größe N für $N = 100$, 1000 und 10000. Zählen Sie die Vergleiche oder messen Sie die tatsächlichen Laufzeiten.

• 6.52 Bestimmen Sie experimentell, ob sich die Abstandsfolge `1 8 23 77 281 1073 4193 16577 ...` für $N = 10000$ verbessern lässt, wenn man einen Abstand hinzufügt oder löscht.

• 6.53 Bestimmen Sie experimentell den Wert von x, der zur niedrigsten Laufzeit für zufällige Dateien führt, wenn man in der Abstandsfolge `1 4 13 40 121 364 1093 3280 9841 ...` für $N = 10000$ die 13 durch x ersetzt.

6.54 Bestimmen Sie experimentell den Wert von α in der Abstandsfolge 1, $\lfloor\alpha\rfloor$, $\lfloor\alpha^2\rfloor$, $\lfloor\alpha^3\rfloor$, $\lfloor\alpha^4\rfloor$, …, der bei zufälligen Dateien mit $N = 10000$ zur geringsten Laufzeit führt.

• 6.55 Ermitteln Sie die 3-Abstandsfolge, die für zufällige Dateien von 1000 Elementen zu möglichst wenigen Vergleichen führt.

•• 6.56 Konstruieren Sie eine Datei mit 100 Elementen, für die Shellsort mit den Abständen `1 8 23 77` zu einer möglichst großen Anzahl von Vergleichen führt.

• 6.57 Zeigen Sie, dass sich jede Zahl größer als oder gleich $(h – 1)(k – 1)$ als Linearkombination (mit nichtnegativen Koeffizienten) von h und k ausdrücken lässt, wenn h und k teilerfremd sind. *Hinweis:* Zeigen Sie, dass h und k einen gemeinsamen Faktor haben müssen, wenn man beliebige zwei der ersten $h – 1$ Vielfachen von k durch h teilt und jeweils der gleiche Rest bleibt.

6.58 Bestimmen Sie experimentell die Werte von h und k, die zu geringsten Laufzeiten für zufällige Dateien führen, wenn man eine Pratt-ähnliche Folge basierend auf h und k verwendet, um 10000 Elemente zu sortieren.

➔

→ 6.59 Die Abstandsfolge `1 5 19 41 109 209 505 929 2161 3905 ...` basiert auf dem Überlagern der Folgen $9*4^i - 9*2^i + 1$ und $4^i - 3*2^i + 1$ für $i > 0$. Vergleichen Sie die Ergebnisse, wenn Sie die Folgen einzeln und in der angegebenen Überlagerung für das Sortieren von 10000 Elementen einsetzen.

6.60 Wir leiten die Abstandsfolge `1 3 7 21 48 112 336 861 1968 4592 13776 ...` ab, indem wir mit einer Basisfolge von teilerfremden Zahlen beginnen (sagen wir `1 3 7 16 41 101`), dann ein Dreieck wie in der Folge von Pratt aufbauen, jetzt aber die i-te Zeile im Dreieck generieren, indem wir das erste Element in der $(i - 1)$-ten Zeile mit dem i-ten Element in der Basisfolge und jedes Element in der $(i - 1)$-ten Zeile mit dem $(i + 1)$-ten Element in der Basisfolge multiplizieren. Bestimmen Sie experimentell eine Basisfolge, die beim Sortieren von 10000 Elementen besser als die angegebene ist.

- 6.61 Führen Sie die Beweise der Eigenschaften 6.7 und 6.8 zu Ende.
- 6.62 Implementieren Sie einen Shellsort, der auf dem Shakersort von Übung 6.40 basiert, und vergleichen Sie ihn mit dem Standardalgorithmus. *Hinweis:* Ihre Abstandsfolge sollte sich grundlegend von der des Standardalgorithmus unterscheiden.

6.9 Sortieren verketteter Listen

Wie wir aus Kapitel 3 wissen, bieten Arrays und verkettete Listen zwei der fundamentalsten Arten, um Daten zu strukturieren, und wir haben eine Implementierung des Verfahrens *Sortieren durch Einfügen* für verkettete Listen als Beispiel einer Listenverarbeitung in Abschnitt 3.4 (Programm 3.10) betrachtet. Die bisher behandelten Sortierimplementierungen gehen alle davon aus, dass die zu sortierenden Daten in einem Array vorliegen und nicht direkt anwendbar sind, wenn wir in einem System arbeiten, das Daten in verketteten Listen organisiert. In bestimmten Fällen können die *Algorithmen* nützlich sein, allerdings nur, wenn sie Daten in der hauptsächlich sequenziellen Art und Weise verarbeiten, die wir effizient für verkettete Listen unterstützen können.

Programm 3.10 ist ein Testprogramm ähnlich Programm 6.1, um Sortieroperationen mit verketteten Listen zu testen. Analog zum Testprogramm für Arrays initialisiert es die Liste, sortiert sie und zeigt ihren Inhalt an. Wie Kapitel 3 erläutert hat, kann man mit verketteten Listen auf einer höheren Abstraktionsebene arbeiten, jedoch können wir uns durch den hier verwendeten Low-Level-Ansatz besser auf die Verbindungsmanipulationen konzentrieren, die die Algorithmen und Datenstrukturen an sich ausmachen und denen unser Hauptinteresse in diesem Buch gilt. Es ist relativ einfach, eine Schnittstelle für Listen zu definieren und generische Implementierungen zu erstellen, wie wir sie für Arrays in Abschnitt 6.2 entwickelt haben (siehe Übung 6.66).

Für die Manipulierung verketteter Strukturen gibt es eine Grundregel, die in vielen Anwendungen kritisch ist, aber nicht immer aus unserem Code hervorgeht. In einer komplexen Umgebung kann es passieren, dass Referenzen auf die Listenknoten, die wir manipulieren, durch andere Teile des Anwendungssystems verwaltet werden (d.h. sie sind in Multilisten eingebunden). Die Möglichkeit, auf Knoten über Referenzen zuzugreifen, die

außerhalb der Sortierroutine verwaltet werden, bedeutet, dass unsere Programme *nur Verbindungen in Knoten ändern und keine Schlüssel oder andere Informationen beeinflussen sollten*. Wenn wir beispielsweise eine Vertauschung vornehmen wollen, scheint es am einfachsten zu sein, die Elemente zu tauschen (wie wir es beim Sortieren von Arrays getan haben). Dann aber findet jede Referenz auf die beiden Knoten über irgendeine andere Verbindung den geänderten Wert vor und hat wahrscheinlich nicht die gewünschte Wirkung. Wir müssen die Verbindungen selbst ändern, und zwar in einer Weise, dass die Knoten in sortierter Reihenfolge erscheinen, wenn wir die Liste über die uns zugänglichen Verbindungen traversieren, ohne deren Reihenfolge zu beeinflussen, wenn der Zugriff auf die Knoten über irgendwelche anderen Verbindungen erfolgt. Dieses Vorgehen führt zwar zu komplizierteren Implementierungen, ist jedoch normalerweise unumgänglich.

Die Sortierverfahren Sortieren durch Einfügen, Sortieren durch Auswählen und Bubblesort können wir auf Implementierungen mit verketteten Listen übertragen, auch wenn jede Variante für sich interessante Herausforderungen bietet.

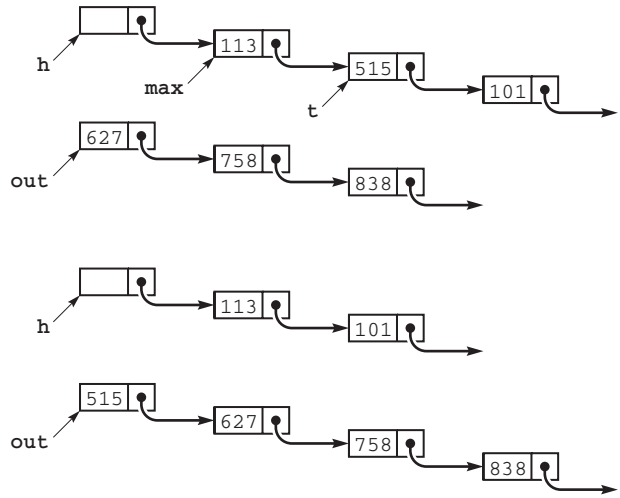


Abbildung 6.16: Sortieren durch Auswählen für verkettete Listen

Diese Zeichnung zeigt einen Schritt beim Sortieren durch Auswählen für verkettete Listen. Wir verwalten eine Eingabeliste, auf die `h.next` *verweist, und eine Ausgabeliste, auf die* `out` *verweist (oben). Wir durchsuchen die Eingabeliste, um* `max` *auf den Knoten vor dem (und* `t` *auf den) Knoten mit dem größten Element zeigen zu lassen. Diese Referenzen benötigen wir, damit wir den* `t`*-Knoten aus der Eingabeliste entfernen (und ihre Länge um 1 verringern) und ihn vor den Anfang der Ausgabeliste stellen (und ihre Länge um 1 erhöhen) können. Die Ausgabeliste bleibt dabei in sortierter Reihenfolge (unten). Wenn wir auf diese Weise fortfahren, haben wir schließlich die Eingabeliste aufgebraucht und die Knoten in der richtigen Reihenfolge in die Ausgabeliste übernommen.*

Sortieren durch Auswählen ist unkompliziert: Wir verwalten eine Eingabeliste (die anfänglich die Daten enthält) und eine Ausgabeliste (die das sortierte Ergebnis sammelt). Dann durchsuchen wir einfach die Eingabeliste, um das größte Element zu ermitteln, ent-

fernen es aus der Liste und fügen es vor dem Anfang der Ausgabeliste ein (siehe Abbildung 6.16). Programm 6.19 zeigt eine Implementierung. Programm 3.10 gibt eine Implementierung des Verfahrens Sortieren durch Einfügen für verkettete Listen an und Übung 6.68 bezieht sich auf Bubblesort. Diese Verfahren eignen sich für das Sortieren kurzer Listen, weisen aber ähnliche Leistungsnachteile auf, wie wir sie für ihre arraybasierten Entsprechungen gefunden haben.

Programm 6.19
Sortieren durch Auswählen für verkettete Listen

Sortieren durch Auswählen für eine verkettete Liste ist unkompliziert, unterscheidet sich aber leicht von der Arrayversion, weil es einfacher ist, ein Element vor dem Anfang einer Liste einzufügen. Wir verwalten eine Eingabeliste (auf die `h.next` verweist) und eine Ausgabeliste (auf die `out` verweist). Solange die Eingabeliste nicht leer ist, durchsuchen wir sie nach dem größten verbleibenden Element, entfernen es dann aus der Eingabeliste und fügen es vor dem Anfang der Ausgabeliste ein. Diese Implementierung verwendet eine Hilfsroutine `findMax`, die eine Verbindung auf den Knoten zurückgibt, dessen Verbindung auf das größte Element in einer Liste verweist.

```
private static Node findMax(Node h)
  {
    for (Node t = h; t.next != null; t = t.next)
      if (h.next.item < t.next.item) h = t;
    return h;
  }
static Node sort(Node h)
  { Node head = new Node(-1, h), out = null;
    while (head.next != null)
      { Node max = findMax(head);
        Node t = max.next; max.next = t.next;
        t.next = out; out = t;
      }
    return out;
  }
```

In manchen Situationen der Listenverarbeitung können wir darauf verzichten, explizit ein Sortierverfahren zu implementieren. Beispielsweise können wir von vornherein die Liste jederzeit sortiert halten und dabei neue Knoten wie beim Sortieren durch Einfügen in die Liste aufnehmen. Dieser Ansatz bedeutet geringe Zusatzkosten, wenn zum Beispiel das Einfügen relativ selten vorkommt oder die Liste klein ist. Aus bestimmten Gründen kann es erforderlich sein, die gesamte Liste zu durchsuchen, bevor neue Knoten eingefügt werden (etwa, um auf Duplikate zu prüfen). In Kapitel 14 behandeln wir einen Algorithmus, der sortierte verkettete Listen verwendet. Außerdem stellen die Kapitel 12 bis 14 eine Reihe von Datenstrukturen vor, die von einer Sortierung der Daten profitieren.

ÜBUNGEN

▷ 6.63 Geben Sie den Inhalt der Eingabeliste und der Ausgabeliste an, wenn Sie Programm 6.19 für die Schlüssel `A S O R T I N G E X A M P L E` ausführen.

6.64 Implementieren Sie ein Clientprogramm zum Testen der Leistung für Sortierverfahren mit verketteten Listen (siehe Übung 6.8).

6.65 Implementieren Sie ein Clientprogramm, um Sortierverfahren mit verketteten Listen für extreme Fälle zu testen (siehe Übung 6.9).

6.66 Entwerfen Sie einen abstrakten Datentyp für sortierbare verkettete Listen, der eine Methode für zufällige Initialisierung, einen Methode zur Initialisierung vom Standardeingabestrom, eine `sort`-Methode und eine Ausgabemethode einbindet. Alle Methoden sollen eine Knotenreferenz als Parameter übernehmen und ebenfalls als Rückgabewert liefern, um den funktionellen Programmierstil von Programm 3.10 zu unterstützen.

6.67 Schreiben Sie eine Klasse, die Ihre abstrakte Klasse von Übung 6.66 erweitert, um verkettete Listen zu implementieren, deren Datensätze Schlüssel vom Typ `double` sind.

6.68 Implementieren Sie Bubblesort für eine verkettete Liste. *Achtung:* Vertauschen zweier benachbarter Elemente in einer verketteten Liste ist schwieriger, als es auf den ersten Blick aussieht.

6.69 Das in Programm 3.10 verwendete Verfahren *Sortieren durch Einfügen* führt dazu, dass das Verfahren *Sortieren durch Einfügen mit verketteter Liste* bei bestimmten Eingabedateien deutlich langsamer läuft als die Arrayversion. Beschreiben Sie eine derartige Datei und erläutern Sie das Problem.

• 6.70 Implementieren Sie eine Version von Shellsort mit verketteter Liste, die für große zufällige Dateien unwesentlich mehr Zeit oder Platz benötigt als die Arrayversion. *Hinweis:* Verwenden Sie Bubblesort.

• 6.71 Entwickeln Sie eine Implementierung mit verketteter Liste für die Array-ADT-Schnittstelle in Programm 6.5, sodass Sie beispielsweise Programm 6.6 einsetzen können, um Sortierimplementierungen mit verketteten Listen zu debuggen.

6.10 Schlüsselindiziertes Zählen

Viele Sortieralgorithmen erreichen einen Leistungsgewinn, indem sie spezielle Eigenschaften der Schlüssel nutzen. Sehen wir uns als Beispiel folgendes Problem an: Sortiere eine Datei von N Elementen, deren Schlüssel unterschiedliche Ganzzahlen zwischen 0

und $N - 1$ sind. Mit der folgenden Anweisung, die ein temporäres Array `b` verwendet, können wir dieses Problem sofort lösen:

```
for (i = 0; i < N; i++) b[a[i]] = a[i];
```

Das heißt, wir sortieren, indem wir die Schlüssel als *Indizes* verwenden und nicht einfach als abstrakte Elemente, die zu vergleichen sind. In diesem Abschnitt betrachten wir ein elementares Verfahren, das Schlüsselindizierung auf diese Weise einsetzt, um effizient zu sortieren, wenn die Schlüssel Ganzzahlen in einem kleinen Bereich sind.

Wenn alle Schlüssel den Wert 0 haben, ist das Sortieren trivial, aber nehmen wir nun an, dass es zwei eindeutige Schlüsselwerte 0 und 1 gibt. Ein derartiges Sortierproblem kann auftreten, wenn wir aus einer Datei die Elemente abtrennen möchten, die einen bestimmten (möglicherweise komplizierten) Eignungstest bestanden haben: Wir ordnen dem Schlüssel 0 die Bedeutung »angenommen« und dem Schlüssel 1 die Bedeutung »zurückgewiesen« zu. Man könnte nun die Anzahl der Nullen zählen und in einem zweiten Durchgang die Elemente aus den Eingabedaten `a` in das temporäre Array `b` übertragen. Dazu verwendet man ein Array mit zwei Zählern wie folgt: Wir initialisieren `cnt[0]` mit 0 und `cnt[1]` mit der Anzahl der 0-Schlüssel in der Datei. Damit zeigen wir an, dass es in der Datei keine Schlüssel kleiner als 0 und `cnt[1]` Schlüssel kleiner als 1 gibt. Natürlich könnten wir das Array `b` füllen, indem wir Nullen in die ersten Felder (beginnend bei `b[cnt[0]]` oder `b[0]`) und Einsen in die Felder ab `b[cnt[1]]` schreiben. Das heißt, der Code

```
for (i = 0; i < N; i++) b[cnt[a[i]]++] = a[i];
```

verteilt die Elemente von `a` nach `b`. Auch hier erhalten wir eine schnelle Sortierung, indem wir die Schlüssel als Indizes verwenden (um Elemente zwischen `cnt[0]` und `cnt[1]` herauszugreifen).

Dieser Ansatz lässt sich sofort verallgemeinern. Ein realistischeres Problem im gleichen Sinne ist folgendes: Sortiere eine Datei von N Elementen, deren Schlüssel Ganzzahlen zwischen 0 und $M - 1$ sind. Wir können die beiden Basismethoden im vorherigen Absatz auf einen Algorithmus namens *schlüsselindizier-*

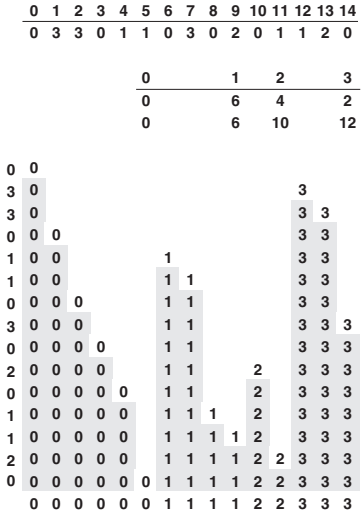


Abbildung 6.17: Sortieren durch schlüsselindiziertes Zählen

Zuerst bestimmen wir, wie viele Schlüssel von jedem Wert in der Datei vorhanden sind. In diesem Beispiel sind es 6 Nullen, 4 Einsen, 2 Zweien und 3 Dreien. Dann bilden wir die Teilsummen, um die Anzahl der Schlüssel, die kleiner als die jeweiligen Schlüssel sind, zu ermitteln: 0 Schlüssel sind kleiner als 0*, 6 Schlüssel sind kleiner als* 1*, 10 Schlüssel sind kleiner als* 2 *und 12 Schlüssel sind kleiner als* 3 *(Tabelle in der Mitte). Dann verwenden wir die Teilsummen als Indizes, um die Schlüssel an ihre Positionen zu bringen: Die* 0 *am Anfang der Datei wird an Position 0 geschrieben; dann inkrementieren wir den Zeiger, der dem Schlüssel* 0 *entspricht, um auf die Position zu zeigen, wo die nächste* 0 *einzutragen ist. Danach wird die* 3 *von der nächsten Position auf der linken Seite in der Datei an die Position 12 gebracht (da es 12 Schlüssel kleiner als* 3 *gibt); ihr korrespondierender Zählerwert wird inkrementiert usw.*

tes Zählen erweitern, der dieses Problem effizient löst, wenn *M* nicht zu groß ist. Genau wie bei zwei Schlüsselwerten besteht die Idee darin, die Anzahl der Schlüssel mit den einzelnen Werten zu zählen und dann in einem zweiten Durchgang durch die Datei anhand der Zählerwerte die Elemente an ihre Positionen zu bringen. Zuerst zählen wir die Anzahl der Schlüssel für jeden Wert, dann berechnen wir die Teilsummen, um jeweils die Anzahl der Schlüssel kleiner oder gleich jedem Wert zu erhalten. Dann verwenden wir genau wie bei zwei Werten diese Zählerwerte als Indizes, um die Schlüssel zu verteilen. Für jeden Schlüssel sehen wir seinen zugehörigen Zählerwert als Index an, der auf das Ende des Schlüsselblocks mit demselben Wert zeigt, verwenden den Index, um den Schlüssel nach `b` zu verteilen, und dekrementieren. Abbildung 6.17 veranschaulicht diesen Vorgang. Eine Implementierung ist in Programm 6.20 angegeben.

Programm 6.20 Schlüsselindiziertes Zählen

Die erste `for`-Schleife initialisiert die Zählerwerte auf 0; die zweite `for`-Schleife setzt den zweiten Zähler auf die Anzahl der Nullen, den dritten Zähler auf die Anzahl der Einsen usw. Dann addiert die dritte `for`-Schleife einfach diese Werte, um die Anzahlen der Schlüssel kleiner oder gleich dem Schlüsselwert zu bilden, der dem Zähler entspricht. Diese Anzahlen geben nun die Indizes für das Ende des jeweiligen Blocks in der Datei an, in den die Schlüssel gehören. Die vierte `for`-Schleife verschiebt die Schlüssel in ein Hilfsarray `b` gemäß dieser Indizes, und die letzte Schleife verschiebt die sortierte Datei zurück nach `a`. Die Schlüssel müssen Ganzzahlen kleiner als `M` sein, damit dieser Code funktioniert. Wir können ihn aber auch leicht modifizieren, um derartige Schlüssel aus komplizierteren Elementen zu extrahieren (siehe Übung 6.75).

```
static void distCount(int a[], int l, int r)
  { int i, j, cnt[] = new int[M];
    int b[] = new int[a.length];
    for (j = 0; j < M; j++) cnt[j] = 0;
    for (i = l; i <= r; i++) cnt[a[i]+1]++;
    for (j = 1; j < M; j++) cnt[j] += cnt[j-1];
    for (i = l; i <= r; i++) b[cnt[a[i]]++] = a[i];
    for (i = l; i <= r; i++) a[i] = b[i-l];
  }
```

Eigenschaft 6.12 *Schlüsselindiziertes Zählen ist ein Sortierverfahren mit linearer Laufzeit, vorausgesetzt, dass der Bereich unterscheidbarer Schlüsselwerte innerhalb eines konstanten Faktors der Dateigröße bleibt.*

Jedes Element wird zweimal bewegt: das erste Mal, um es zu verteilen, das zweite Mal, um es zurück in das Originalarray zu bringen. Außerdem wird jeder Schlüssel zweimal referenziert: das erste Mal, um die Anzahl der Schlüsselwerte zu ermitteln, das zweite Mal, um die Verteilung durchzuführen. Die beiden anderen `for`-Schleifen im Algorithmus ermitteln die Zählerwerte und tragen nur unwesentlich zur Laufzeit bei, sofern die Anzahl der Zähler nicht deutlich größer als die Dateigröße wird. ■

Wenn sehr große Dateien zu sortieren sind, kann das Hilfsarray `b` zu einem Speicherproblem führen. Programm 6.20 lässt sich modifizieren, um das In-situ-Sortieren durchzuführen (sodass kein Hilfsarray erforderlich ist). Dieses Verfahren ist eng mit den elementaren Verfahren verwandt, die wir in späteren Kapiteln behandeln, sodass wir dieses Thema auf die Übungen 10.19 und 10.20 in Abschnitt 10.3 verschieben. Wie Kapitel 10 zeigt, gehen diese Platzeinsparungen zu Lasten der Stabilitätseigenschaften des Algorithmus und schränken somit seine Anwendbarkeit ein, weil Anwendungen mit einer großen Anzahl doppelter Schlüssel oftmals andere zugehörende Schlüssel haben, deren relative Reihenfolge zu bewahren ist. Ein besonders wichtiges Beispiel für eine derartige Anwendung lernen wir in Kapitel 10 kennen.

ÜBUNGEN

○ 6.72 Geben Sie eine spezialisierte Version des schlüsselindizierten Zählens für das Sortieren von Dateien an, deren Elemente nur einen von drei Werten (*a*, *b* oder *c*) annehmen können.

6.73 Angenommen, wir verwenden Sortieren durch Einfügen für eine zufällig geordnete Datei, in der die Elemente nur einen von drei Werten haben können. Ist die Laufzeit linear, quadratisch oder liegt sie irgendwo dazwischen?

▷ 6.74 Zeigen Sie, wie schlüsselindiziertes Zählen die Datei `A B R A C A D A B R A` sortiert.

6.75 Implementieren Sie schlüsselindiziertes Zählen für Elemente, die möglicherweise große Datensätze mit ganzzahligen Schlüsseln aus einem kleinen Bereich sind.

6.76 Geben Sie eine Implementierung für schlüsselindiziertes Zählen an, die die Elemente im Eingabearray `a` nicht verschiebt, sondern ein zweites Array `b` erstellt, sodass `b[0]` der Index des Elements in `a` ist, das als erstes Elemente in der sortierten Folge erscheint, `b[1]` der Index des Elements in `a` ist, das als zweites Element erscheint, usw.