Kapitel 8

Elementare Sortierverfahren

Als ersten Einstieg in das Gebiet der Sortieralgorithmen wollen wir einige »elementare« Methoden untersuchen, die für kleine Dateien oder Dateien mit einer speziellen Struktur geeignet sind. Es gibt verschiedene Gründe dafür, diese einfachen Sortieralgorithmen recht ausführlich zu betrachten. Erstens bieten sie einen relativ mühelosen Weg, Terminologie und grundlegende Mechanismen für Sortieralgorithmen kennenzulernen, so daß wir einen geeigneten Hintergrund für das Studium der komplizierteren Algorithmen schaffen. Zweitens ist es bei einem großen Teil der Anwendungen von Sortierverfahren besser, diese einfachen Methoden zu benutzen als die leistungsfähigeren Mehrzweckverfahren. Schließlich lassen sich einige der einfachen Methoden zu besseren Mehrzweckverfahren weiterentwickeln, oder sie können verwendet werden, um die Effizienz leistungsfähigerer Verfahren zu erhöhen.

Wie bereits erwähnt wurde, gibt es verschiedene Sortier-Anwendungen, bei denen ein relativ einfacher Algorithmus die zu bevorzugende Methode sein kann. Sortierprogramme werden oft nur einmal (oder nur wenige Male) benutzt. Wenn die Anzahl der zu sortierenden Elemente nicht zu groß ist (sagen wir, weniger als fünfhundert Elemente), so kann es durchaus effizienter sein, lediglich eine einfache Methode anzuwenden, als ein kompliziertes Verfahren zu implementieren und zu debuggen. Elementare Verfahren sind für kleine Dateien (mit weniger als etwa fünfzig Elementen) immer geeignet; ein komplizierter Algorithmus dürfte für eine kleine Datei kaum gerechtfertigt sein, es sei denn, daß eine sehr große Anzahl solcher Dateien zu sortieren ist. Weitere Typen von Dateien, die sich relativ leicht sortieren lassen, sind solche, die bereits nahezu sortiert (oder bereits sortiert!) sind, oder solche, die eine große Anzahl von gleichen Schlüsseln enthalten. Bei solchen gut strukturierten Dateien sind einfache Methoden weit besser geeignet als Mehrzweckverfahren.

In der Regel benötigen die elementaren Methoden, die wir betrachten werden, etwa N^2 Schritte, um N zufällig angeordnete Elemente zu sortieren. Falls N hinreichend klein ist, ist das im allgemeinen kein Problem, und falls die Elemente nicht zufällig angeordnet sind, können einige der Verfahren weit schneller ablaufen als kompliziertere. Es muß jedoch betont werden, daß diese Verfahren für umfangreiche, zufällig angeordnete Dateien *nicht* benutzt werden sollten, mit der bemerkenswerten Ausnah-

me von Shellsort, welches tatsächlich das bevorzugte Sortierverfahren für sehr viele Anwendungen ist.

Spielregeln

Bevor wir einige spezifische Algorithmen betrachten, wird es von Nutzen sein, einige allgemeine Termini und grundlegende Annahmen für Sortieralgorithmen zu erörtern. Wir werden Verfahren zum Sortieren von *Dateien* (files) mit *Datensätzen* (records) betrachten, welche Schlüssel (keys) enthalten. Die Schlüssel, die nur ein Teil der Datensätze sind (oft ein kleiner Teil), werden verwendet, um das Sortieren zu steuern. Die Aufgabe des Sortierverfahrens besteht darin, die Datensätze so umzuordnen, daß ihre Schlüssel gemäß einer gewissen klar definierten Ordnung (gewöhnlich nach der numerischen oder alphabetischen Reihenfolge) geordnet sind.

Wenn die zu sortierende Datei im Speicher untergebracht werden kann (oder, unter unseren Bedingungen, wenn sie in einem Feld (**array**) von C++ gespeichert werden kann), so spricht man von einer *internen* Sortiermethode. Das Sortieren von Dateien auf einem Magnetband oder einer Magnetplatte wird *externes* Sortieren genannt. Der Hauptunterschied zwischen den beiden Typen besteht darin, daß bei einem internen Sortierverfahren ein leichter Zugriff auf jeden Datensatz möglich ist, während ein externes Sortierverfahren den Zugriff auf die Datensätze sequentiell oder zumindest in großen Blöcken realisieren muß. In Kapitel 13 werden wir einige externe Sortierverfahren betrachten, doch die meisten der von uns behandelten Algorithmen betreffen interne Sortiermethoden.

Wie üblich, ist der wichtigste Parameter der Leistungsfähigkeit, der uns interessiert, die Laufzeit unserer Sortieralgorithmen. Die ersten vier Methoden, die wir in diesem Kapitel betrachten, benötigen für das Sortieren von N Elementen eine Zeit, die proportional zu N^2 ist, während weiterentwickelte Verfahren N Elemente in einer Zeit sortieren können, die zu $N \log N$ proportional ist. (Es läßt sich zeigen, daß kein Sortieralgorithmus mit weniger als $N \log N$ Vergleichen zwischen Schlüsseln auskommt.) Nach der Untersuchung der einfachen Methoden betrachten wir eine weiterentwickelte Methode, die in einer zu $N^{3/2}$ proportionalen Zeit oder schneller ablaufen kann, und wir werden sehen, daß es Verfahren gibt, die digitale Eigenschaften von Schlüsseln ausnutzen und dadurch eine Gesamtlaufzeit erzielen, die proportional zu N ist.

Der zusätzliche Speicherplatz, der durch einen Sortieralgorithmus belegt wird, ist der zweite wichtige Faktor, den wir berücksichtigen müssen. Grundsätzlich lassen sich die Verfahren in drei Typen einteilen: Verfahren, die am Ort sortieren und keinen zusätzlichen Speicherplatz benötigen, außer vielleicht für einen kleinen Stapel oder eine kleine Tabelle; Verfahren, bei denen eine Darstellung mittels verketteter Liste benutzt wird, so daß sie im Speicher N zusätzliche Worte für Listenzeiger benötigen;

sowie Verfahren, die genügend zusätzlichen Speicherplatz benötigen, um eine weitere Kopie des zu sortierenden Feldes zu speichern.

Ein weiteres Merkmal von Sortierverfahren, welches in der Praxis manchmal von Bedeutung ist, ist die *Stabilität*. Ein Sortierverfahren wird *stabil* genannt, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält. Wenn zum Beispiel eine alphabetisch geordnete Liste von Studenten eines Kurses nach deren Noten sortiert wird, so liefert eine stabile Methode eine Liste, in der Studenten mit der gleichen Note nach wie vor in alphabetischer Reihenfolge aufgeführt werden; eine nichtstabile Methode erzeugt dagegen im allgemeinen eine Liste, in der von der alphabetischen Reihenfolge keine Spur mehr vorhanden ist. Die meisten einfachen Methoden sind stabil, die meisten der bekannten komplizierten Algorithmen jedoch nicht. Wenn die Stabilität von Bedeutung ist, so kann sie erzwungen werden, indem vor dem Sortieren jeder Schlüssel mit einem kleinen Index versehen wird oder auf irgendeine andere Weise verlängert wird. Stabilität wird leicht als gegeben angenommen; Anwender reagieren oft mit Verwunderung auf die unangenehmen Effekte der Instabilität. In Wirklichkeit gewährleisten nur wenige Methoden Stabilität, ohne wesentlich mehr Zeit oder Speicherplatz zu benötigen.

Das folgende Programm dient dem Ziel, die allgemeinen Konventionen zu illustrieren, die wir benutzen werden. Es besteht aus einem Hauptprogramm, welches N Zahlen einliest und dann ein Unterprogramm aufruft, um sie zu sortieren. Im vorliegenden Beispiel sortiert das Unterprogramm nur die ersten drei eingelesenen Zahlen; das entscheidende ist, daß anstelle von `sort3` jedes beliebige Sortierprogramm eingesetzt und von diesem »Treiber«-Programm aufgerufen werden könnte.

```
inline void swap(itemType a[], int i, int j)
  { itemType t = a[i]; a[i] = a[j]; a[j] = t; }
sort3(itemType a[], int N)
  {
    if (a[1] > a[2]) swap(a, 1, 2);
    if (a[1] > a[3]) swap(a, 1, 3);
    if (a[2] > a[3]) swap(a, 2, 3);
  }
const int maxN = 100;
main()
  {
    int N, i; itemType v, a[maxN+1];
    N = 0; while (cin >> v) a[++N] = v;
    a[0] = 0;
    sort3(a, N);
    for (i = 1; i <= N; i++) cout << a[i] << ' ';
    cout << '\n';
  }
```

Wie in Kapitel 3 konzentrieren wir uns auf algorithmische Fragen und lassen den Typ der zu sortierenden Objekte als `itemType` undefiniert. Dabei tun wir so, als arbeite-

ten wir mit Algorithmen, die einfach Felder von ganzen Zahlen der Größe nach oder Buchstaben in alphabetischer Reihenfolge sortieren. In C++ ist es sehr einfach, mittels `typedef` oder `template` solche Algorithmen für den Einsatz in praktischen Anwendungen anzupassen, in denen umfangreiche Schlüssel oder Datensätze auftreten.

Normalerweise realisieren Sortierprogramme den Zugriff auf Datensätze auf zweierlei Weise: Entweder es erfolgt ein Zugriff auf Schlüssel zum Zwecke des Vergleichs, oder ein Zugriff auf ganze Datensätze, um sie zu verlagern. Die Mehrzahl der Algorithmen, die wir untersuchen werden, können unter Verwendung dieser beiden Operationen für beliebige Datensätze formuliert werden. Wenn die zu sortierenden Datensätze groß sind, sollte man normalerweise vermeiden, sie umherzuschieben, indem man ein »indirektes Sortieren« vornimmt: Hierbei werden die Datensätze selbst nicht notwendigerweise umgeordnet, sondern vielmehr wird ein Feld von Zeigern (oder Indizes) so umgeordnet, daß der erste Zeiger auf den kleinsten Datensatz zeigt usw. Die Schlüssel können entweder mit den Datensätzen (wenn sie umfangreich sind) oder mit den Zeigern (wenn sie klein sind) gespeichert werden. Falls erforderlich, können die Datensätze dann nach dem Sortieren umgeordnet werden, wie weiter unten in diesem Kapitel beschrieben wird.

Die Prozedur `swap` realisiert eine »Austausch«-Operation; sie ist als `inline` deklariert, da sie in vielen Sortierprogrammen und dort oft in der innersten Schleife auftritt. Tatsächlich benutzt das Programm eine sogar noch stärker eingeschränkte Variante des Zugriffs auf die Datei: Es handelt sich um drei Anweisungen der Form »vergleiche zwei Datensätze und tausche sie, falls erforderlich, aus, so daß derjenige mit dem kleineren Schlüssel vorn steht«. Programme, die auf solche Anweisungen beschränkt sind, sind deswegen von Interesse, weil sie für eine hardwaremäßige Implementation gut geeignet sind. Wir werden uns dieser Frage in Kapitel 40 ausführlicher zuwenden.

Wir wollen nicht näher auf die allgemeine Frage der »Bündelungsprobleme« von Sortier-Routinen eingehen, aber trotzdem einige Hinweise darauf geben, wie die Möglichkeiten von C++ zum sinnvollen Einsatz der von uns untersuchten Algorithmen in konkreten Anwendung genutzt werden können. Beispielsweise erwähnten wir bereits die Möglichkeit, mittels `typedef` oder `template` den Algorithmus für ein Vielzahl anderer Schlüsseltypen verfügbar zu machen. Ein weiteres Beispiel: In C++ ist es sinnvoll, das Feld der Sortier-Routine als Parameter zu übergeben, während es in einer anderen Sprache möglicherweise nicht sinnvoll ist. Sollte das Programm stattdessen mit einem globalen Feld operieren? Sollte die Sortier-Routine Bestandteil einer Klasse sein, welche die üblichen Feld-Operationen um das Sortieren erweitert? Einige Betriebssysteme machen es recht einfach, einfache Programme wie das obige, die als »Filter« zwischen ihren Eingaben und ihren Ausgaben dienen, zusammenzusetzen. Andererseits sind solche Mechanismen für viele Anwendungen nicht wirklich notwendig. Offensichtlich gelten diese Bemerkungen für viele andere Algorithmen, die wir untersuchen werden, wenn auch die genannten Effekte vielleicht bei Sortieralgorithmen am deutlichsten sichtbar werden.

Wir fügen auch nicht viel »Fehlerprüfungs«-Code ein, obwohl das bei Anwendungen normalerweise ratsam ist. Zum Beispiel sollte die Treiber-Routine wahrscheinlich prüfen, daß `N` nicht größer ist als `maxN` (und `sort3` sollte prüfen, daß `N = 3` gilt).

Eine andere nützliche Prüfung wäre, mit Hilfe des Treibers zu sichern, daß das Feld nach dem Aufruf von `sort3` sortiert ist. Dies ist keine Garantie dafür, daß das Programm funktioniert (warum?), doch es kann helfen, Fehler zu finden.

Manche Programme verwenden einige weitere globale Variablen. Deklarationen, die nicht offensichtlich sind, werden mit in den Programmcode aufgenommen. Weiterhin werden wir gewöhnlich `a[0]` (und manchmal `a[N+1]`) reservieren, um spezielle Schlüssel zu speichern, die von einigen der Algorithmen verwendet werden. Oft werden wir für Beispiele anstelle von Zahlen Buchstaben benutzen; diese werden in der offensichtlichen Weise unter Verwendung der in C++ vorhandenen standardmäßigen Umwandlungsfunktionen zwischen ganzen Zahlen und Buchstaben behandelt.

Selection Sort

Einer der einfachsten Sortieralgorithmen läuft wie folgt ab: Finde zuerst das kleinste Element im Feld und tausche es gegen das an der ersten Stelle befindliche Element aus, finde danach das zweitkleinste Element und tausche es gegen das an zweiter Stelle befindliche Element aus und fahre in dieser Weise fort, bis das gesamte Feld sortiert ist. Diese Methode wird *Selection Sort* (Sortieren durch (direktes) Auswählen) genannt, da sie darin besteht, daß wiederholt das kleinste verbliebene Element »ausgewählt« wird, wie in Abbildung 8.1 dargestellt ist. Der erste Durchlauf hat keine Wirkung, weil im Feld kein Element existiert, das kleiner als das links stehende A ist. Beim zweiten Durchlauf ist das zweite A das kleinste verbliebene Element, daher wird es gegen das an zweiter Stelle stehende S ausgetauscht. Danach wird beim dritten Durchlauf das erste E gegen das an dritter Stelle befindliche O ausgetauscht, danach beim vierten Durchlauf das zweite E gegen das an vierter Stelle stehende R usw.

Das folgende Programm ist eine Implementation dieses Prozesses. Für jedes `i` von `1` bis `N - 1` tauscht es `a[i]` gegen das kleinste Element in `a[i],...,a[N]` aus:

```
void selection(itemType a[], int N)
  {
    int i, j, min;
    for (i = 1; i < N; i++)
      {
        min = i;
        for (j = i+1; j <= N; j++)
            if (a[j] < a[min]) min = j;
        swap(a, min, i);
      }
  }
```

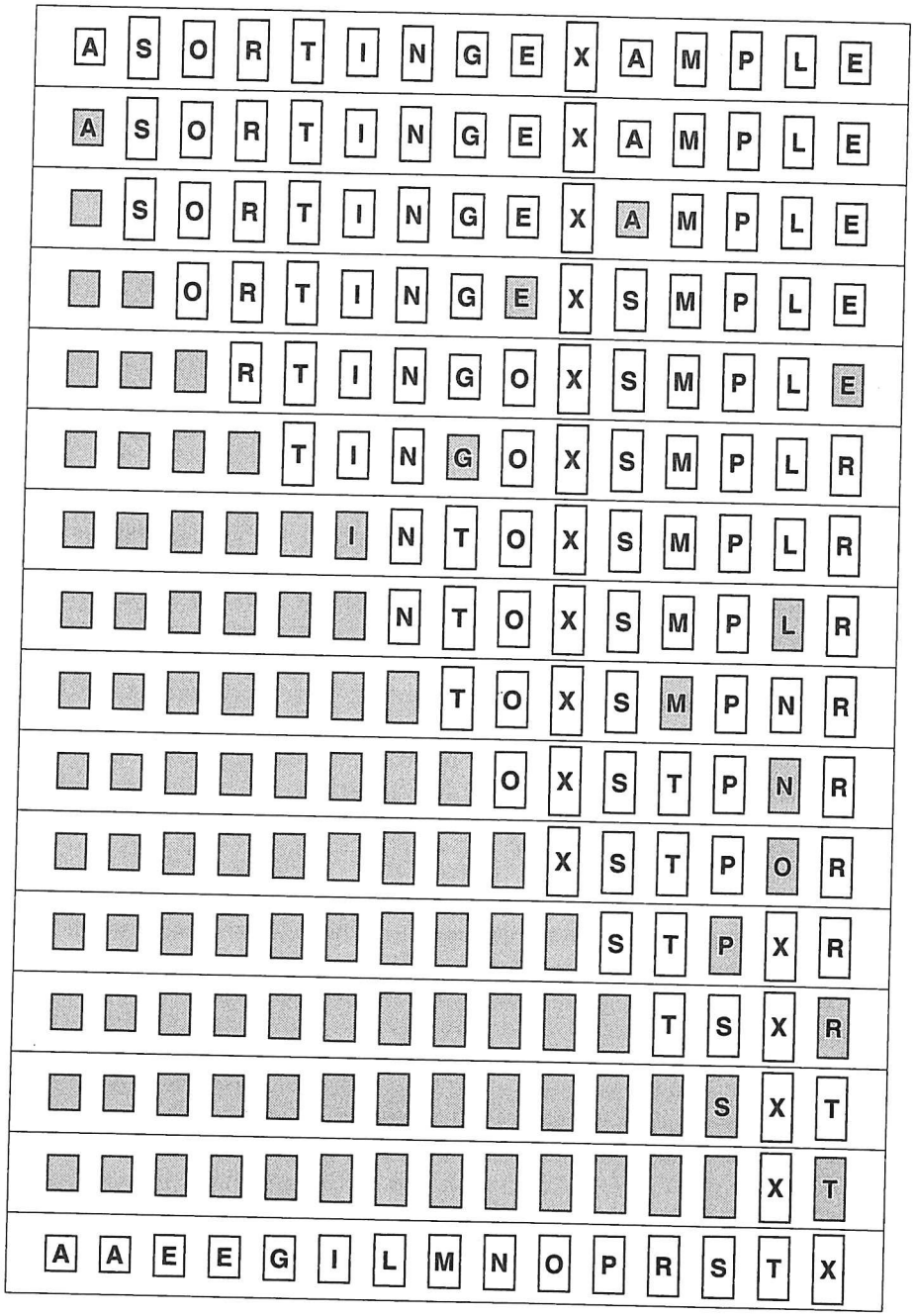
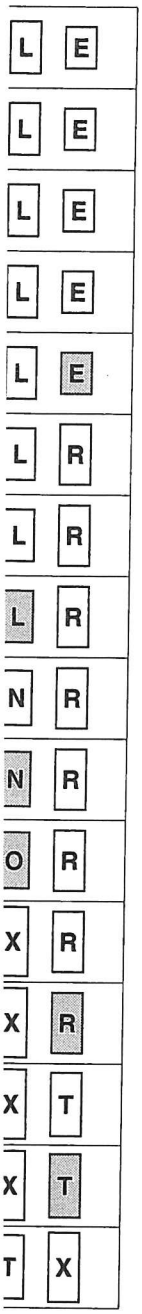


Abbildung 8.1 Selection Sort.



Während der Index `i` von links nach rechts durch die Datei wandert, befinden sich die Elemente links vom Index auf ihrer endgültigen Position im Feld (und werden nicht wieder berührt), so daß das Feld vollständig sortiert ist, wenn der Index das rechte Ende erreicht.

Dies ist eine der einfachsten Sortiermethoden, und sie ist für kleine Dateien sehr gut brauchbar. Die »innere Schleife« ist der Vergleich `a[j] < a[min]` (plus der Code, der benötigt wird, um `j` zu inkrementieren und zu überprüfen, ob es `N` nicht übersteigt), der kaum einfacher sein könnte. Weiter unten erörtern wir, wie oft diese Anweisungen voraussichtlich ausgeführt werden müssen.

Weiterhin hat Selection Sort, obwohl es offenbar eine »gewaltsame« Verfahrensweise ist, tatsächlich eine sehr wichtige Anwendung: Da jedes Element wirklich höchstens einmal bewegt wird, ist Selection Sort die bevorzugte Methode, um Dateien mit sehr großen Datensätzen und kleinen Schlüsseln zu sortieren. Dies wird noch im einzelnen betrachtet.

Insertion Sort

Ein Algorithmus, der fast genauso einfach wie Selection Sort, aber vielleicht flexibler ist, ist *Insertion Sort* (Sortieren durch (direktes) Einfügen). Dies ist die Methode, die Menschen oft beim Kartenspiel anwenden, um ihre Karten zu sortieren: Betrachte die Elemente eines nach dem anderen und füge jedes an seinen richtigen Platz zwischen den bereits betrachteten ein (wobei diese sortiert bleiben). Das gerade betrachtete Element wird eingefügt, indem die größeren Elemente einfach um eine Position nach rechts bewegt werden und das Element dann auf dem freigewordenen Platz eingefügt wird, wie Abbildung 8.2 zeigt. Das S auf der zweiten Position ist größer als das A, daher muß es nicht bewegt werden. Wenn das O auf der dritten Position vorgefunden wird, wird es gegen das S ausgetauscht, um A O S in eine sortierte Reihenfolge zu bringen usw.

Dieser Prozeß ist im folgenden Programm implementiert. Für jedes `i` von `2` bis `N` werden die Elemente `a[1],...,a[i]` sortiert, indem `a[i]` an die entsprechende Stelle in der sortierten Liste von Elementen in `a[1],...,a[i-1]` gesetzt wird:

```
void insertion(itemType a[], int N)
  {
    int i, j; itemType v;
    for (i = 2; i <= N; i++)
      {
        v = a[i]; j = i;
        while (a[j-1] > v)
          { a[j] = a[j-1]; j--; }
        a[j] = v;
      }
  }
```

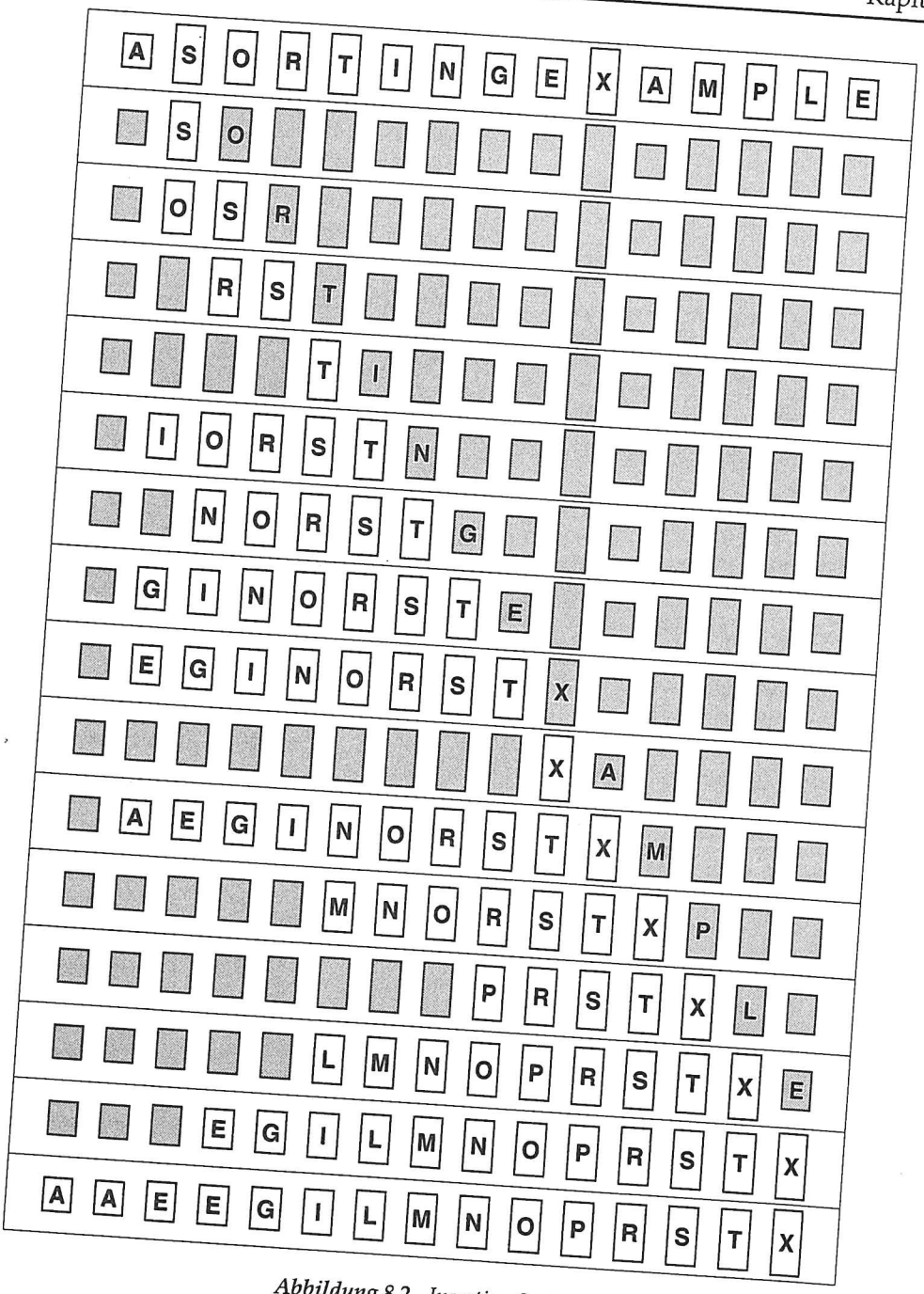
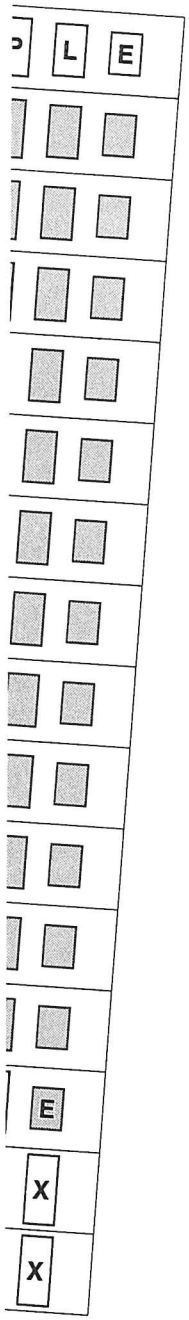


Abbildung 8.2 *Insertion Sort.*



Wie bei Selection Sort sind die Elemente links vom Zeiger `i` während des Sortierens in einer sortierten Reihenfolge angeordnet, doch sie befinden sich nicht auf ihrer endgültigen Position, da sie möglicherweise bewegt werden müssen, um für später gefundene kleinere Elemente Platz zu machen. Das Feld ist jedoch vollständig sortiert, wenn der Zeiger das rechte Ende erreicht.

Es muß noch eine wichtige Einzelheit betrachtet werden: Für die meisten Eingaben funktioniert die Prozedur `insertion` nicht! Die `while`-Anweisung läuft über das linke Ende des Feldes hinaus, wenn *v* das kleinste Element im Feld ist. Um Abhilfe zu schaffen, setzen wir einen »Marken«-Schlüssel (»sentinel«-key) auf `a[0]`, den wir mindestens so klein wählen wie das kleinste Element im Feld. Marken werden gewöhnlich in Situationen wie dieser verwendet, um die Aufnahme eines nahezu immer positiv ausfallenden Tests (im vorliegenden Falle `j>1`) in die innere Schleife zu vermeiden.

Falls es aus irgendeinem Grund schwierig ist, eine Marke zu verwenden (zum Beispiel ist der kleinste Schlüssel vielleicht nicht auf einfache Weise definiert), könnte der Test `while j>1 && a[j-1]>v` benutzt werden. Dies ist ungünstig, da `j=1` nur selten zutrifft, weshalb sollten wir also diesen Test häufig in der inneren Schleife ausführen? Wir bemerken, daß, wenn `j` den Wert `1` hat, der obige Test `a[j-1]` nicht erreichen kann, auf Grund der Art und Weise, wie in C++ logische Ausdrücke ausgewertet werden – manche andere Sprachen könnten in einem solchen Fall einen Zugriff zum Feld außerhalb der Grenzen realisieren. Ein anderer Weg, diese Situation in C++ zu meistern, besteht darin, eine `break`- oder `goto`-Anweisung zu benutzen, die aus der `while`-Schleife herausführt. (Manche Programmierer ziehen es vor, einigen Aufwand in Kauf zu nehmen, um `goto`-Anweisungen zu vermeiden, zum Beispiel indem sie innerhalb der Schleife Vorkehrungen treffen, um zu gewährleisten, daß die Schleife abbricht. Im vorliegenden Fall scheint eine solche Lösung kaum gerechtfertigt zu sein, da sie das Programm nicht klarer macht und jedesmal über die Schleife Ballast hinzufügt, nur um für ein seltenes Ereignis abgesichert zu sein.)

Exkurs: Bubble Sort

Ein elementares Sortierverfahren, das in einführenden Kursen oft gelehrt wird, ist *Bubble Sort* (Sortieren durch (direktes) Austauschen): Durchlaufe immer wieder die Datei und vertausche jedesmal, wenn es notwendig ist, benachbarte Elemente; wenn bei einem Durchlauf kein Austausch mehr erforderlich ist, ist die Datei sortiert. Eine Implementation des Verfahrens wird nachfolgend angegeben.

```
void bubble(itemType a[], int N)
  {
    int i, j;
    for (i = N; i >= 1; i--)
      for (j = 2; j <= i; j++)
        if (a[j-1] > a[j]) swap(a, j-1, j);
  }
```

Man muß etwas nachdenken, um sich davon zu überzeugen, daß dieser Weg überhaupt zum Ziel führt. Hierzu bemerken wir, daß jedesmal, wenn während des ersten Durchlaufs das maximale Element vorgefunden wird, dieses mit jedem der rechts von ihm befindlichen Elemente vertauscht wird, und dies so lange, bis es die Position am rechten Ende des Feldes erreicht hat. Während des zweiten Durchlaufs wird dann das zweitgrößte Element an die richtige Position bewegt usw. Somit läuft Bubble Sort wie eine Abart des Selection Sort ab, obwohl viel mehr Aufwand getrieben wird, um jedes Element an die richtige Position zu bringen.

Kenngrößen der Leistungsfähigkeit elementarer Sortiermethoden

Die Abbildungen 8.3, 8.4 und 8.5 dienen zur direkten Veranschaulichung der Merkmale der Arbeitsweise von Selection Sort, Insertion Sort und Bubble Sort. Diese Abbildungen zeigen den Inhalt des Feldes `a` für jeden der Algorithmen, nachdem die äußere Schleife $N/4$, $N/2$ und $3N/4$ mal durchlaufen worden ist (beginnend mit einer zufälligen Permutation der ganzen Zahlen 1 bis N als Eingabe). In den Diagrammen wurde für `a[i]=j` ein Quadrat an die Stelle (i,j) gesetzt. Ein ungeordnetes Feld entspricht damit einer zufälligen Anordnung von Quadraten; in einem sortierten Feld erscheint jedes Quadrat oberhalb des links von ihm befindlichen Quadrates. Zur Klarheit der Abbildungen zeigen wir Permutationen (zufällige Anordnungen der ganzen Zahlen 1 bis N), für die nach dem Sortieren die Quadrate alle längs der Hauptdiagonale angeordnet sind. Die Bilder zeigen, wie sich die verschiedenen Methoden auf dieses Ziel hin bewegen.

Abbildung 8.3 zeigt, wie sich Selection Sort von links nach rechts bewegt, wobei es Elemente in ihre endgültige Position bringt, ohne »zurückzublicken«. Was aus diesem Bild nicht ersichtlich ist, ist die Tatsache, daß bei Selection Sort die meiste Zeit dafür aufgewandt wird, das minimale Element in dem »unsortierten« Teil des Feldes zu suchen.

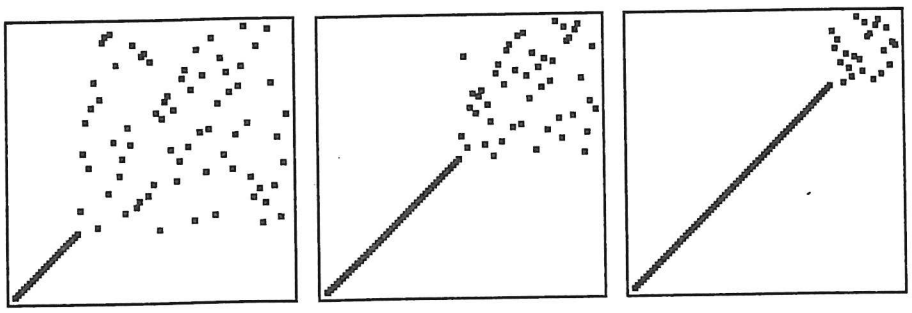


Abbildung 8.3 *Sortieren einer zufälligen Permutation durch Selection Sort.*

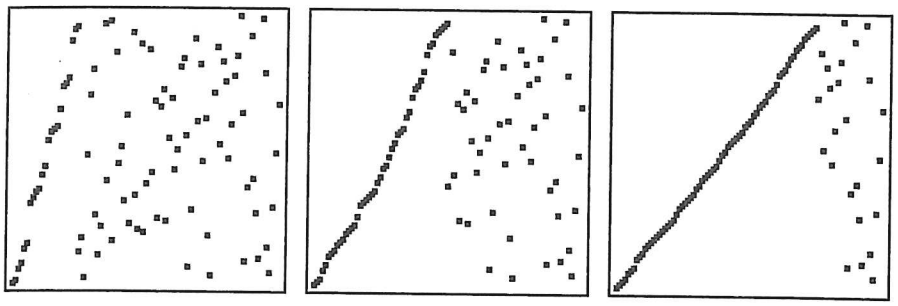


Abbildung 8.4 *Sortieren einer zufälligen Permutation durch Insertion Sort.*

Abbildung 8.4 zeigt, wie sich Insertion Sort gleichfalls von links nach rechts bewegt, wobei neu vorgefundene Elemente an der entsprechenden Stelle eingefügt werden, ohne daß weiter nach vorn geschaut wird. Der linke Teil des Feldes verändert sich ständig.

Abbildung 8.5 zeigt die Ähnlichkeit zwischen Selection Sort und Bubble Sort. Bei Bubble Sort wird auf jeder Stufe das größte verbliebene Element ausgewählt, doch es wird einige Mühe dafür »vergeudet«, im »unsortierten« Teil des Feldes eine gewisse Ordnung herzustellen.

Alle Verfahren sind sowohl im ungünstigsten als auch im durchschnittlichen Fall quadratisch und erfordern keinen zusätzlichen Speicherplatz. Daher sind Vergleiche zwischen ihnen von der Länge der inneren Schleifen oder von speziellen Merkmalen der Eingabedaten abhängig.

Eigenschaft 8.1 *Selection Sort benötigt ungefähr $N^2/2$ Vergleiche und N Austauschoperationen.*

Diese Eigenschaft läßt sich leicht einsehen, wenn man Abbildung 8.1 betrachtet, bei der es sich um eine Tabelle von der Größe $N \times N$ handelt, in der ein Buchstabe jeweils einem Vergleich entspricht. Doch dies betrifft etwa die Hälfte der Elemente, und zwar

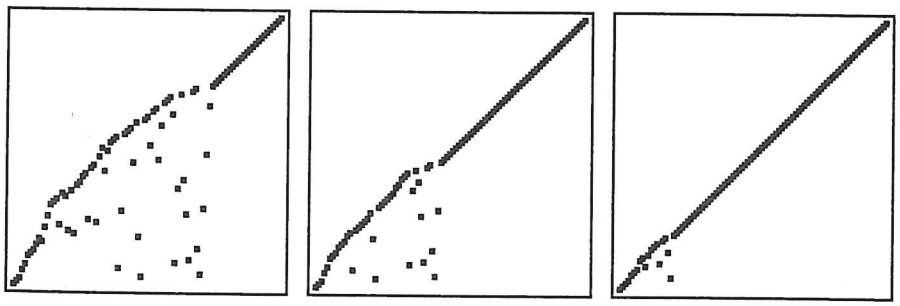


Abbildung 8.5 *Sortieren einer zufälligen Permutation durch Bubble Sort.*

diejenigen oberhalb der Diagonalen. Die $N - 1$ Elemente auf der Diagonalen (mit Ausnahme des letzten) entsprechen jeweils einem Austausch. Genauer: Für jedes i von 1 bis $N - 1$ erfolgen ein Austausch und $N - i$ Vergleiche, wodurch sich eine Gesamtzahl von $N - 1$ Austauschoperationen und $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2$ Vergleichen ergibt. Diese Beobachtungen gelten unabhängig davon, welche Eingabedaten vorliegen; der einzige Teil von Selection Sort, der von den Eingabedaten abhängt, ist die Anzahl, wie oft `min` einen neuen Wert zugewiesen bekommt. Im ungünstigsten Fall könnte dies ebenfalls eine quadratische Funktion von N sein, im durchschnittlichen Fall erweist es sich jedoch, daß diese Größe nur $O(N \log N)$ ist, so daß zu erwarten ist, daß die Laufzeit von Selection Sort relativ unempfindlich gegenüber Änderungen der Eingabedaten ist. ■

Eigenschaft 8.2 *Insertion Sort benötigt im Durchschnitt ungefähr $N^2/4$ Vergleiche und $N^2/8$ Austauschoperationen, im ungünstigsten Fall doppelt so viele.*

Wie aus der obigen Implementation ersichtlich ist, ist die Anzahl der Vergleiche und der »halben Austausche« (Bewegungen) die gleiche. Diese Größe läßt sich leicht anhand Abbildung 8.2 veranschaulichen, der $N \times N$ - Tabelle, die die Arbeitsweise des Algorithmus im einzelnen zeigt. In diesem Falle sind die Elemente unterhalb der Diagonalen zu zählen, im ungünstigsten Fall alle. Für zufällige Eingabedaten ist zu erwarten, daß sich jedes Element im Durchschnitt etwa den halben Weg zurück bewegt, so daß die Hälfte der Elemente unterhalb der Diagonalen zu zählen wäre. (Es ist nicht schwierig, diese Überlegungen exakter zu formulieren.) ■

Eigenschaft 8.3 *Bubble Sort benötigt im Durchschnitt und im ungünstigsten Fall ungefähr $N^2/2$ Vergleiche und $N^2/2$ Austauschoperationen.*

Im ungünstigsten Fall (Datei in umgekehrter Reihenfolge) ist es klar, daß der i-te Durchlauf von Bubble Sort $N - i$ Vergleiche *und* Austauschoperationen erfordert, so daß der Beweis wie für Selection Sort geführt werden kann. Jedoch ist die Laufzeit bei Bubble Sort von den Eingabedaten abhängig. Wir sehen zum Beispiel, daß nur ein Durchlauf benötigt wird, wenn die Datei bereits geordnet ist (Insertion Sort ist in diesem Falle auch schnell). Es zeigt sich, daß - wie oben erwähnt - die Leistungsfähigkeit im durchschnittlichen Fall nicht wesentlich besser ist als im ungünstigsten Fall; diese Analyse ist jedoch wesentlich komplizierter. ■

Eigenschaft 8.4 *Insertion Sort ist für »fast sortierte« Dateien linear.*

Obwohl der Begriff einer »fast sortierten« Datei notwendigerweise recht ungenau ist, läßt sich Insertion Sort gut auf gewisse nichtzufällige Dateien anwenden, die in der Praxis oft auftreten. Gewöhnlich werden für solche Anwendungen unnötigerweise Mehrzweck-Sortierverfahren benutzt; in Wirklichkeit kann Insertion Sort aus der in der Datei vorhandenen Ordnung Nutzen ziehen.

Betrachten wir zum Beispiel den Ablauf von Insertion Sort für eine Datei, die bereits sortiert ist. Jedes Element wird sofort als an seinem richtigen Platz in der Datei befindlich erkannt, und die Gesamtlaufzeit ist linear. Das gleiche gilt für Bubble Sort. Selection Sort jedoch ist nach wie vor quadratisch. Sogar dann, wenn eine Datei nicht

vollständig sortiert ist, kann Insertion Sort recht zweckmäßig sein, da seine Laufzeit stark von der in der Datei vorhandenen Ordnung abhängt. Die Laufzeit ist von der Anzahl der *Inversionen* abhängig: Für jedes Element sind dazu die links von ihm befindlichen Elemente zu zählen, die größer sind. Dies ist der Abstand, um den die Elemente bewegt werden müssen, wenn sie bei Insertion Sort in die Datei eingefügt werden. Eine Datei, in der eine gewisse Ordnung vorhanden ist, weist weniger Inversionen auf als eine zufällig zusammengestellte Datei.

Nehmen wir an, zu einer sortierten Datei sollen einige Elemente hinzugefügt werden, so daß eine größere sortierte Datei erzeugt wird. Ein Weg zur Realisierung besteht darin, die neuen Elemente an das Ende der Datei anzuhängen und dann einen Sortieralgorithmus aufzurufen. Es ist klar, daß die Anzahl der Inversionen in einer solchen Datei gering ist: Eine Datei, in der sich nur eine konstante Zahl von Elementen nicht an ihrem Platz befindet, weist nur eine lineare Anzahl von Inversionen auf. Ein anderes Beispiel ist eine Datei, in der jedes Element sich nur in einem gewissen konstanten Abstand von seiner endgültigen Position befindet. Dateien dieser Art können in den Anfangsstadien einiger weiterentwickelter Sortierverfahren erzeugt werden; zu einem gewissen Zeitpunkt lohnt es sich dann, zu Insertion Sort überzuwechseln.

Für derartige Dateien ist Insertion Sort in Bezug auf die Leistung sogar den komplizierten Verfahren überlegen, die in den nachfolgenden Kapiteln beschrieben werden. ■

Um den Vergleich der Methoden weiterzuführen, ist es erforderlich, die Kosten von Vergleichs- und Austauschoperationen zu analysieren, ein Faktor, der wiederum von der Größe der Datensätze und Schlüssel abhängt. Wenn zum Beispiel die Datensätze wie in den obigen Implementationen aus einem Wort bestehende Schlüssel sind, so dürfte ein Austausch (zwei Zugriffe auf das Feld) etwa doppelt so teuer sein wie ein Vergleich. In einer solchen Situation sind die Laufzeiten von Selection Sort und Insertion Sort, grob gesagt, vergleichbar, während Bubble Sort doppelt so viel Zeit benötigt. (Tatsächlich dürfte Bubble Sort unter nahezu allen Umständen halb so schnell sein wie Insertion Sort!) Falls jedoch die Datensätze im Vergleich zu den Schlüsseln groß sind, ist Selection Sort am besten geeignet.

Eigenschaft 8.5 *Für Dateien mit großen Datensätzen und kleinen Schlüsseln ist Selection Sort linear.*

Nehmen wir an, daß die Kosten für einen Vergleich 1 Zeiteinheit und die Kosten für einen Austausch M Zeiteinheiten betragen. (Dies könnte zum Beispiel der Fall sein, wenn die Datensätze aus M Worten und die Schlüssel aus einem Wort bestehen.) Dann benötigt Selection Sort etwa N^2 Zeiteinheiten für Vergleiche und etwa NM Zeiteinheiten für Austauschoperationen, um eine Datei *der Größe* NM zu sortieren. Falls $N = O(M)$ gilt, ist das eine lineare Funktion der Datenmenge. ■