# Algorithmen und Datenstrukturen

Bearbeitet von
Thomas Ottmann, Peter Widmayer

6., durchgesehene Auflage 2017. Buch. XXIV, 774 S. Hardcover
ISBN 978 3 662 55649 8
Format (B x L): 17,7 x 24,6 cm
Gewicht: 1546 g

# Kapitel 2

# Sortieren

Untersuchungen von Computerherstellern und -nutzern zeigen seit vielen Jahren, dass mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt. Es ist daher nicht erstaunlich, dass große Anstrengungen unternommen wurden möglichst effiziente Verfahren zum Sortieren von Daten mithilfe von Computern zu entwickeln. Das gesammelte Wissen über Sortierverfahren füllt inzwischen Bände. Noch immer erscheinen neue Erkenntnisse über das Sortieren in wissenschaftlichen Fachzeitschriften (vgl. z. B. [120]) und zahlreiche theoretisch und praktisch wichtige Probleme im Zusammenhang mit dem Problem eine Menge von Daten zu sortieren sind ungelöst. Zunächst wollen wir das Sortierproblem genauer fixieren: Wir nehmen an, es sei eine Menge von *Sätzen* gegeben; jeder Satz besitzt einen *Schlüssel*. Zwischen Schlüsseln ist eine Ordnungsrelation „$<$“ oder „$\leq$“ erklärt. Außer der Schlüsselkomponente können Sätze weitere Komponenten als „eigentliche“ Information enthalten. Wenn nichts Anderes gesagt ist, werden wir annehmen, dass die Schlüssel ganzzahlig sind. Das ist auch ein in der Praxis oft vorliegender Fall. Man denke etwa an Artikelnummern, Hausnummern, Personalnummern, Scheckkartennummern usw. (Es kommen aber auch andere Schlüssel vor, z. B. alphabetisch und insbesondere lexikografisch geordnete Namen!)

Das *Sortierproblem* kann präziser wie folgt formuliert werden. Gegeben ist eine Folge von Sätzen (englisch: items) $s_1, \ldots, s_N$; jeder Satz $s_i$ hat einen Schlüssel $k_i$. Man finde eine Permutation $\pi$ der Zahlen von 1 bis $N$ derart, dass die Umordnung der Sätze gemäß $\pi$ die Schlüssel in aufsteigende Reihenfolge bringt:

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(N)}.$$

In dieser Problemformulierung sind viele Details offen gelassen, die für die Lösung durchaus wichtig sind. Was heißt es, dass eine Folge von Sätzen „gegeben“ ist? Ist damit gemeint, dass sie in schriftlicher Form oder auf Magnetband, Platte, Diskette oder CDROM vorliegen? Ist die Anzahl der Sätze bekannt? Ist das Spektrum der vorkommenden Schlüssel bekannt? Welche Operationen sind erlaubt um die Permutation $\pi$ zu bestimmen? Wie geschieht die „Umordnung“? Sollen die Datensätze physisch bewegt werden oder genügt es eine Information zu berechnen, die das Durchlaufen, insbesondere die Ausgabe der Datensätze in aufsteigender Reihenfolge erlaubt?

Wir können unmöglich alle in der Realität auftretenden Parameter des Sortierproblems berücksichtigen. Vielmehr wollen wir uns auf einige prinzipielle Aspekte des


T. Ottmann und P. Widmayer, *Algorithmen und Datenstrukturen*, DOI 10.1007/978-3-662-55650-4_2

Problems beschränken und dazu weitere Annahmen machen. Zunächst nehmen wir an, dass die Menge der zu sortierenden Datensätze vollständig im Hauptspeicher Platz findet. Sortierverfahren, die von dieser Annahme ausgehen, heißen auch *interne* Sortierverfahren. Im Abschnitt 2.7 dieses Kapitels gehen wir dann auch auf *externe* Sortierverfahren ein, die eine auf einem externen Speichermedium (Diskette, Platte, Band) vorliegende Folge von Datensätzen voraussetzen, die nicht gänzlich im Hauptspeicher Platz findet.

Die Lösung des Sortierproblems verlangt in jedem Fall die Lösung zweier Teilprobleme, nämlich

(a) eines Informationsbeschaffungsproblems und

(b) eines Datentransportproblems.

**Zu (a):** Meistens nimmt man an, dass als einzige verwendbare Information zur Lösung des Sortierproblems, also zur Bestimmung der Permutation $\pi$, das Ergebnis von Vergleichen zwischen je zwei Schlüsseln zugelassen ist. Wir können also feststellen, ob für zwei Schlüssel $k$ und $k'$ gilt

$$k = k',\ k < k' \text{ oder } k > k'.$$

Wir gehen von dieser Annahme in den Abschnitten 2.1 bis 2.4 und 2.6 bis 2.8 aus. Für ganzzahlige Schlüssel ist es aber natürlich auch sinnvoll, andere Operationen wie Addition, Subtraktion, Multiplikation und ganzzahlige Division zuzulassen. Wir gehen darauf im Abschnitt 2.5 ein.

**Zu (b):** In der Regel verlangt man, dass als Lösung des Sortierproblems die zu sortierenden Datensätze in einem zusammenhängenden Speicherbereich nach aufsteigenden Schlüsseln geordnet vorliegen sollen. Dazu müssen sie bewegt werden, wenn sie nicht schon von vornherein so vorlagen. Als „Bewegungen“ lässt man unter anderem zu: Das Vertauschen zweier benachbarter oder zweier beliebiger Sätze; das Verschieben einzelner Sätze oder ganzer Blöcke um eine feste oder beliebige Distanz; das Platzieren an eine direkt oder indirekt gegebene Speicheradresse.

Eine Folge von $N$ im Hauptspeicher, also intern, vorliegenden Sätzen mit ganzzahligen Schlüsseln kann man programmtechnisch einfach als Array der Länge $N$ realisieren.

```
type
  item = record
           key : integer;
           info : { infotype }
         end;
  sequence = array [1 .. N] of item;
var
  a : sequence
```

Eine Lösung des Sortierproblems (für eine intern gegebene Folge von Datensätzen) besteht dann in der Angabe einer Prozedur, die das als Eingabe- und Ausgabeparameter

übergebene Array $a$ verändert. Es soll erreicht werden, dass nach Aufruf der Prozedur die Elemente von $a$ nach aufsteigenden Schlüsseln sortiert sind. D. h. für alle $i$, $1 \leq i < N$, gilt

$$a[i].key \leq a[i+1].key.$$

Wir setzen also für alle internen Sortierverfahren folgenden einheitlichen Rahmen voraus (außer bei einigen rekursiv formulierten Sortierverfahren, wo die Parameterliste anders angegeben ist).

```
program Rahmen_für_Sortierverfahren (input, output);
const
  N = {Anzahl der zu sortierenden Sätze};
type
  item = record
           key: integer;
           info: { infotype }
         end;
  sequence = array [0 .. N] of item;
var
  a : sequence;
procedure XYZ-sort (var a : sequence);
  {hier folgt die jeweilige Sortierprozedur}
begin
  {Eingabe: Lies a[1], ..., a[N]};
  XYZ-sort(a);
  {Ausgabe: Schreibe a[1], ..., a[N]}
end.
```

Dass der Array-Typ *sequence* hier mit Index 0 beginnend indiziert ist, hat lediglich programmtechnische Gründe. Die zu sortierenden Elemente stehen nach wie vor an den Positionen 1 bis $N$.

Wir behandeln im Abschnitt 2.1 *elementare Sortierverfahren* (Sortieren durch Auswahl, Sortieren durch Einfügen, Shellsort und Bubblesort). Für diese Verfahren ist typisch, dass im ungünstigsten Fall $\Theta(N^2)$ Vergleichsoperationen zwischen Schlüsseln ausgeführt werden müssen um $N$ Schlüssel zu sortieren.

Das im Abschnitt 2.2 behandelte Verfahren *Quicksort* benötigt im Mittel nur $O(N \log N)$ Vergleichsoperationen; es ist das interne Sortierverfahren mit der besten mittleren Laufzeit, weil es im Detail sehr effizient implementiert werden kann.

Zwei Verfahren, die eine Menge von $N$ Schlüsseln stets mit nur $O(N \log N)$ Vergleichsoperationen zu sortieren erlauben, sind die Verfahren *Heapsort* und *Mergesort*, die in den Abschnitten 2.3 und 2.4 diskutiert werden. Im Abschnitt 2.8 zeigen wir, dass $\Omega(N \log N)$ Vergleichsoperationen auch tatsächlich notwendig sein können. Im Abschnitt 2.5 lassen wir die Voraussetzung fallen, dass nur Vergleichsoperationen zwischen Schlüsseln zugelassen werden. Wir geben ein Verfahren (*Radixsort*) an, das die arithmetischen Eigenschaften der zu sortierenden Schlüssel ausnutzt.

# 2.1 Elementare Sortierverfahren

Wir gehen in diesem Abschnitt davon aus, dass die zu sortierenden $N$ Datensätze Elemente eines global vereinbarten Feldes $a$ sind. Es wird jeweils eine Sortierprozedur mit $a$ als Eingabe- und Ausgabeparameter angegeben, die bewirkt, dass nach Ausführung der Prozedur die ersten $N$ Elemente von $a$ so angeordnet sind, dass die Schlüsselkomponenten aufsteigend sortiert sind:

$$a[1].key \leq a[2].key \leq \ldots \leq a[N].key$$

Wir erläutern vier verschiedene Verfahren. *Sortieren durch Auswahl* folgt der nahe liegenden Idee, die Sortierung durch Bestimmung des Elements mit kleinstem, zweitkleinstem, drittkleinstem,... usw. Schlüssel zu erreichen. Die jedem Skatspieler geläufige Methode des Einfügens des jeweils nächsten Elements an die richtige Stelle liegt dem Verfahren *Sortieren durch Einfügen* zu Grunde. Das von D. L. Shell vorgeschlagene Verfahren *Shellsort*, vgl. [183], kann als Verbesserung des Sortierens durch Einfügen angesehen werden. *Bubblesort* ist ein Sortierverfahren, das solange zwei jeweils benachbarte, nicht in der richtigen Reihenfolge stehende Elemente vertauscht, bis keine Vertauschungen mehr nötig sind, das Feld $a$ also sortiert ist. Wir geben in jedem Fall zunächst eine verbale Beschreibung des Sortierverfahrens an und bringen dann eine mögliche Implementation in Pascal.

Zur Messung der Laufzeit der Verfahren verwenden wir zwei nahe liegende Größen. Dies ist zum einen die Anzahl der zum Sortieren von $N$ Sätzen ausgeführten *Schlüsselvergleiche* und zum anderen die Anzahl der ausgeführten *Bewegungen* von Datensätzen. Für beide Parameter interessieren uns die im günstigsten Fall (best case), die im schlechtesten Fall (worst case) und die im Mittel (average case) erforderlichen Anzahlen. Wir bezeichnen die jeweiligen Größen mit

$$C_{min}(N),\ C_{max}(N) \text{ und } C_{mit}(N)$$

für die Anzahlen der Schlüsselvergleiche (englisch: comparisons) und mit

$$M_{min}(N),\ M_{max}(N) \text{ und } M_{mit}(N)$$

für die Anzahlen der Bewegungen (englisch: movements). Die Mittelwerte $C_{mit}(N)$ und $M_{mit}(N)$ werden dabei üblicherweise auf die Menge aller $N!$ möglichen Ausgangsanordnungen von $N$ zu sortierenden Datensätzen bezogen.

## 2.1.1 Sortieren durch Auswahl

**Methode:** Man bestimme diejenige Position $j_1$, an der das Element mit minimalem Schlüssel unter $a[1],\ldots,a[N]$ auftritt und vertausche $a[1]$ mit $a[j_1]$. Dann bestimme man diejenige Position $j_2$, an der das Element mit minimalem Schlüssel unter $a[2],\ldots,a[N]$ auftritt (das ist das Element mit zweitkleinstem Schlüssel unter allen $N$ Elementen), und vertausche $a[2]$ mit $a[j_2]$ usw., bis alle Elemente an ihrem richtigen Platz stehen.

Wir bestimmen also der Reihe nach das *i-kleinste Element*, $i = 1,\ldots,N-1$, und setzen es an die richtige Position. Genauer gesagt bestimmen wir natürlich die Position, an der das Element mit dem $i$-kleinsten Schlüssel steht. D. h. für jedes $i = 1,\ldots,N-1$ gehen wir der Reihe nach wie folgt vor. Wir können voraussetzen, dass $a[1],\ldots,a[i-1]$ bereits die $i-1$ kleinsten Schlüssel in aufsteigender Reihenfolge enthält. Wir suchen unter den verbleibenden $N-i+1$ Elementen das mit kleinstem Schlüssel, sagen wir $a[min]$, und vertauschen $a[i]$ und $a[min]$.

**Beispiel:** Gegeben sei ein Feld mit sieben Schlüsseln:

| $j$ : | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $a[j].key$ : | 15 | 2 | 43 | 17 | 4 | 8 | 47 |

Das kleinste Element steht an Position 2; Vertauschen von $a[1]$ und $a[2]$ ergibt:

| $a[j].key$ : | 2 | 15 | 43 | 17 | 4 | 8 | 47 |
|---|---|---|---|---|---|---|---|

Das zweitkleinste Element steht jetzt an Position 5; Vertauschen von $a[2]$ und $a[5]$ ergibt:

| $a[j].key$ : | 2 | 4 | 43 | 17 | 15 | 8 | 47 |
|---|---|---|---|---|---|---|---|

Die weiteren vier Schritte (Bestimmung des $i$-kleinsten Elements und jeweils Vertauschen mit $a[i]$) kann man kurz wie folgt zusammenfassen:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $i = 3$ | : | 2 | 4 | 8 | 17 | 15 | 43 | 47 |
| $i = 4$ | : | 2 | 4 | 8 | 15 | 17 | 43 | 47 |

Ab jetzt $(i = 5,6)$ verändert sich das Feld $a$ nicht mehr.

Man sieht an diesem Beispiel, dass keine Vertauschung nötig ist, wenn das $i$-kleinste Element bereits an Position $i$ steht. Die folgende Pascal-Version des Verfahrens nutzt diese Möglichkeit nicht aus.

```
procedure Auswahlsort (var a : sequence);
  var
    i, j, min : integer;
    t : item; {Hilfsspeicher}
  begin
    for i := 1 to N − 1 do
      begin
        {bestimme Position min des kleinsten unter
         den Elementen a[i],...,a[N]}
        min := i;
```

```
        for j := i + 1 to N do
{*}       if a[j].key < a[min].key
            then min := j;
        {vertausche Elemente an Position i und Position min}
{**}    t := a[min];
        a[min] := a[i];
        a[i] := t
      end
end
```

**Analyse:** Man sieht, dass zur Bestimmung des $i$-kleinsten Elements, $i = 1, \ldots, N-1$, jeweils die Position des Minimums in der Restfolge $a[i], \ldots, a[N]$ bestimmt wird. Die dabei ausgeführte Anzahl von Schlüsselvergleichen (in Programmzeile $\{*\}$) ist unabhängig von der Ausgangsanordnung jeweils $(N-i)$. Damit ist

$$C_{min}(N) = C_{max}(N) = C_{mit}(N) = \sum_{i=1}^{N-1} (N-i) = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = \Theta(N^2).$$

Zählt man nur die Bewegungen von Datensätzen, die in den drei Programmzeilen ab $\{**\}$ ausgeführt werden, so werden, wieder unabhängig von der Ausgangsanordnung, genau

$$M_{min}(N) = M_{max}(N) = M_{mit}(N) = 3(N-1)$$

Bewegungen durchgeführt. Die Abschätzung der in der auf $\{*\}$ folgenden Programmzeile zur Adjustierung des Wertes von *min* ausgeführten Zuweisungen hängt natürlich vom Ergebnis des vorangehenden Schlüsselvergleiches und damit von der Ausgangsanordnung ab. Im günstigsten Fall wird diese Zuweisung nie, im ungünstigsten Fall jedes Mal und damit insgesamt wieder $\Theta(N^2)$ Mal durchgeführt. Wir haben einfache Zuweisungen deswegen getrennt von Schlüsselvergleichen und Bewegungen von Datensätzen betrachtet, weil sie weniger aufwändig sind und in allen unseren Beispielen ohnehin von den Schlüsselvergleichen dominiert werden. Die Abschätzung der mittleren Anzahl von Zuweisungen, die in der auf $\{*\}$ folgenden Zeile ausgeführt werden, ist schwieriger und wird hier übergangen.

Kann man das Verfahren effizienter machen, etwa dadurch, dass man eine „bessere“ Methode zur Bestimmung des jeweiligen Minimums (in der Restfolge) verwendet? Der folgende Satz zeigt, dass dies jedenfalls dann nicht möglich ist, wenn als einzige Operation Vergleiche zwischen Schlüsseln zugelassen sind.

**Satz 2.1** *Jeder Algorithmus zur Bestimmung des Minimums von N Schlüsseln, der allein auf Schlüsselvergleichen basiert, muss wenigstens* $N-1$ *Schlüsselvergleiche ausführen.*

**Beweis:** Jeder Algorithmus zur Bestimmung des Minimums von $N$ Schlüsseln $k_1, \ldots, k_N$ lässt sich als ein nach dem K.-O.-System ausgeführter Wettkampf auffassen: Von zwei Teilnehmern $k_i$ und $k_j$, $1 \le i, j \le N$, $i \ne j$, scheidet der größere aus. Der Sieger des Wettkampfs steht erst dann fest, wenn alle anderen Teilnehmer ausgeschieden sind. Weil bei jedem Wettkampf genau ein Teilnehmer ausscheidet, benötigt man also $N-1$ Wettkämpfe zur Ermittlung des Siegers. □

Obwohl es Sortierverfahren gibt, die mit weniger als $\Theta(N^2)$ Vergleichsoperationen zwischen Schlüsseln auskommen um $N$ Datensätze zu sortieren, ist das Verfahren Sortieren durch Auswahl unter Umständen das bessere. Sind Bewegungen von Datensätzen besonders teuer, aber Vergleichsoperationen zwischen Schlüsseln billig, so ist Sortieren durch Auswahl gut, weil es nur linear viele Bewegungen von Datensätzen ausführt. Dieser Fall kann z. B. für Datensätze mit (kleinem) ganzzahligem Schlüssel, aber umfangreichem und kompliziert strukturiertem Datenteil vorliegen.

## 2.1.2 Sortieren durch Einfügen

**Methode:** Die $N$ zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils bereits sortierte, anfangs leere Teilfolge an der richtigen Stelle eingefügt.

Nehmen wir also an, $a[1],\ldots,a[i-1]$ seien bereits sortiert, d. h. $a[1].key \leq \ldots \leq a$[i-1].*key*. Dann wird das $i$-te Element $a[i]$ an der richtigen Stelle in die Folge $a[1],\ldots,a[i-1]$ eingefügt. Das geschieht so, dass man $a[i].key$ der Reihe nach mit $a[i-1].key, a[i-2].key, \ldots$ vergleicht und das Element $a[j]$ dabei jeweils um eine Position nach rechts verschiebt, für $j = i-1, i-2, \ldots$, wenn $a[j].key > a[i].key$ ist. Sobald man erstmals an eine Position $j$ gekommen ist, sodass $a[j].key \leq a[i].key$ ist, hat man die richtige Stelle gefunden, an der das Element $a[i]$ eingefügt werden kann, nämlich die Position $j+1$.

Das Einfügen des $i$-ten Elementes $a[i]$ an der richtigen Stelle in der Folge der Elemente $a[1],\ldots,a[i-1]$ verlangt also im Allgemeinen ein Verschieben von $j$ Elementen um eine Position nach rechts, wobei $j$ zwischen 0 und $i-1$ liegen kann. Zur Bestimmung der Einfügestelle wird stets eine Vergleichsoperation mehr als die Anzahl der Verschiebungen durchgeführt.

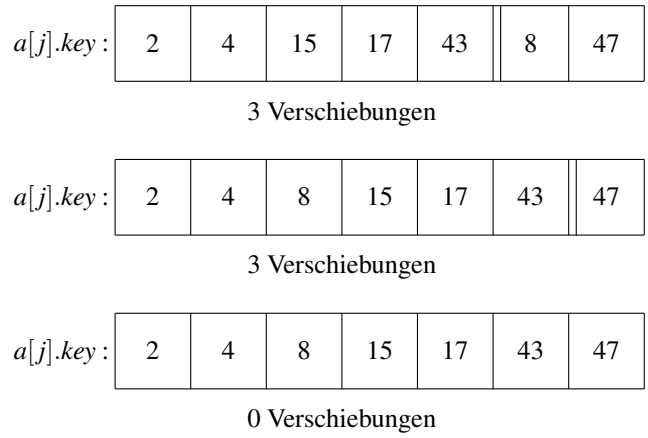
**Beispiel:** Betrachten wir wieder das Feld $a$ mit den sieben Schlüsseln 15, 2, 43, 17, 4, 8, 47. Die aus nur einem einzigen Element bestehende Teilfolge $a[1]$ ist natürlich bereits sortiert. Einfügen von $a[2]$ in diese Folge verlangt die Verschiebung von $a[1]$ um eine Position nach rechts und liefert:

| $j$ : | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $a[j].key$ : | 2 | 15 | 43 | 17 | 4 | 8 | 47 |

Wir haben hier das Ende des bereits sortierten Anfangsstücks des Feldes $a$ durch einen Doppelstrich markiert. Einfügen von $a[3]$ erfordert keine Verschiebung. Einfügen von $a[4]$ bewirkt die Verschiebung von $a[3]$ um eine Position nach rechts und liefert:

| $a[j].key$ : | 2 | 15 | 17 | 43 | 4 | 8 | 47 |
|---|---|---|---|---|---|---|---|

Die weiteren Schritte lassen sich kurz wie folgt angeben:




Es liegt nahe das Verfahren wie folgt in Pascal zu implementieren:

```
procedure Einfügesort (var a : sequence);
var
  i, j, k : integer;
  t : item; {Hilfsspeicher}
begin
  for i := 2 to N do
    begin
    {füge a[i] an der richtigen Stelle in a[1],...,a[i-1] ein}
      j := i;
{**}  t := a[i];
      k := t.key;
{*}   while a[j-1].key > k do
        begin
          {nach rechts verschieben}
{**}      a[j] := a[j-1];
          j := j-1
        end;
{**}  a[j] := t
    end
end
```

Eine genaue Betrachtung des Programms zeigt, dass die **while**-Schleife nicht korrekt terminiert. Ist der Schlüssel $k$ des nächsten einzufügenden Elements $a[i]$ kleiner als die Schlüssel aller Elemente $a[1],\ldots,a[i-1]$, so wird die Bedingung $a[j-1].key > k$ nie falsch für $j = i,\ldots,2$. Eine ganz einfache Möglichkeit, ein korrektes Terminieren der Schleife zu sichern, besteht darin, am linken Ende des Feldes einen Stopper für die lineare Suche abzulegen. Setzt man $a[0].key := k$ direkt vor der **while**-Schleife, wird die Schleife korrekt beendet, ohne dass in Programmzeile $\{*\}$ jedes Mal geprüft werden muss, ob $j$ noch im zulässigen Bereich liegt. Wir gehen im Folgenden von dieser Annahme aus.

**Analyse:** Zum Einfügen des $i$-ten Elementes werden offenbar mindestens ein und höchstens $i$ Schlüsselvergleiche in Programmzeile $\{*\}$ und zwei oder höchstens $i+1$ Bewegungen von Datensätzen in Programmzeilen $\{**\}$ ausgeführt. Daraus ergibt sich sofort

$$C_{min}(N) = N-1; \qquad C_{max}(N) = \sum_{i=2}^{N} i = \Theta(N^2);$$

$$M_{min}(N) = 2(N-1); \qquad M_{max}(N) = \sum_{i=2}^{N} (i+1) = \Theta(N^2).$$

Die im Mittel zum Einfügen des $i$-ten Elementes ausgeführte Anzahl der Schlüsselvergleiche und Bewegungen von Datensätzen hängt offenbar eng zusammen mit der erwarteten Anzahl von Elementen, die im Anfangsstück $a[1],\ldots,a[i-1]$ in der falschen Reihenfolge bezüglich des $i$-ten Elements stehen. Man nennt diese Zahl die erwartete Anzahl von *Inversionen* (Fehlstellungen), an denen das $i$-te Element beteiligt ist.

Genauer: Ist $k_1,\ldots,k_N$ eine gegebene Permutation $\pi$ von $N$ verschiedenen Zahlen, so heißt ein Paar $(k_i,k_j)$ eine Inversion, wenn $i<j$, aber $k_i>k_j$ ist. Die Gesamtzahl der Inversionen einer Permutation $\pi$ heißt *Inversionszahl* von $\pi$. Sie kann als Maß für die „Vorsortiertheit“ von $\pi$ verwendet werden. Sie ist offenbar 0 für die aufsteigend sortierte Folge und $\sum_{i=1}^{N}(N-i) = \Theta(N^2)$ für die absteigend sortierte Folge. Im Mittel kann man erwarten, dass die Hälfte der dem $i$-ten Element $k_i$ vorangehenden Elemente größer als $k_i$ ist. Die mittlere Anzahl von Inversionen und damit die mittlere Anzahl von Schlüsselvergleichen und Bewegungen von Datensätzen, die die Prozedur *Einfügesort* ausführt, ist damit von der Größenordnung

$$\sum_{i=1}^{N} \frac{i}{2} = \Theta(N^2).$$

Wir werden später (im Abschnitt 2.6) Sortierverfahren kennen lernen, die die mit der Inversionszahl gemessene Vorsortierung in einem noch zu präzisierenden Sinne optimal nutzen.

Es ist nahe liegend zu versuchen, das Sortieren durch Einfügen dadurch zu verbessern, dass man ein besseres Suchverfahren zur Bestimmung der Einfügestelle für das $i$-te Element verwendet. Das hilft aber nur wenig. Nimmt man beispielsweise das im Kapitel 3 besprochene binäre Suchen anstelle des in der angegebenen Prozedur benutzten linearen Suchens, so kann man zwar die Einfügestelle mit $\log i$ Schlüsselvergleichen bestimmen, muss aber immer noch im schlechtesten Fall $i$ und im Mittel $i/2$ Bewegungen von Datensätzen ausführen um für das $i$-te Element Platz zu machen. Ein wirklich besseres Verfahren zum Sortieren von $N$ Datensätzen, das auf der dem Sortieren durch Einfügen zu Grunde liegenden Idee basiert, erhält man dann, wenn man die zu sortierenden Sätze in einer ganz anderen Datenstruktur (nicht als Array) speichert. Es gibt Strukturen, die das Einfügen eines Elementes mit einer Gesamtschrittzahl (das ist mindestens die Anzahl von Schlüsselvergleichen und Bewegungen) erlauben, die proportional zu $\log d$ ist; dabei ist $d$ der Abstand der aktuellen Einfügestelle von der jeweils vorangehenden. Wir besprechen ein darauf gegründetes Sortierverfahren (Sortieren durch *lokales Einfügen*) im Abschnitt 2.6.

### 2.1.3 Shellsort

**Methode:** An Stelle des beim Sortieren durch Einfügen benutzten wiederholten Verschiebens um eine Position nach rechts versuchen wir die Elemente in größeren Sprüngen schneller an ihre endgültige Position zu bringen. Zunächst wählen wir eine abnehmende und mit 1 endende Folge von so genannten Inkrementen $h_t, h_{t-1}, \ldots, h_1$. Das ist eine Folge positiv ganzzahliger Sprungweiten, z. B. die Folge 5, 3, 1. Dann betrachten wir der Reihe nach für jedes der abnehmenden Inkremente $h_i$, $t \geq i \geq 1$, alle Elemente im Abstand $h_i$ voneinander. Man kann also die gegebene Folge auffassen als eine Menge von (höchstens) $h_i$ verzahnt gespeicherten, logisch aber unabhängigen Folgen $f_j$, $1 \leq j \leq h_i$. Die zur Folge $f_j$ gehörenden Elemente stehen im Feld an Positionen $j, j+h_i, j+2h_i, \ldots$, also an Positionen $j+m \cdot h_i$, $0 \leq m \leq \lfloor \frac{(N-j)}{h_i} \rfloor$. Anfangs haben wir $h_t$ solcher Folgen, später, bei $h_1 = 1$, gerade eine einzige Folge. Für jedes $h_i$, $t \geq i \geq 1$, sortieren wir jede Folge $f_j$, $1 \leq j \leq h_i$, mittels Einfügesort. Weil das in $f_j$ zu einem Folgenelement benachbarte Element um $h_i$ Positionen versetzt im Feld gespeichert ist, bewirkt dieses Sortierverfahren, dass ein Element bei einer Bewegung um $h_i$ Positionen nach rechts wandert. Der letzte dieser $t$ Durchgänge ist identisch mit dem gewöhnlichen Einfügesort; nur müssen die Elemente jetzt nicht mehr so weit nach rechts verschoben werden, da sie vorher schon ein gutes Stück gewandert sind.

Diese auf D. L. Shell zurückgehende Methode nennt man auch *Sortieren mit abnehmenden Inkrementen*. Man nennt eine Folge $k_1, \ldots, k_N$ von Schlüsseln *h-sortiert*, wenn für alle $i$, $1 \leq i \leq N-h$, gilt: $k_i \leq k_{i+h}$. Für eine abnehmende Folge $h_t, \ldots, h_1 = 1$ von Inkrementen wird also mithilfe von Sortieren durch Einfügen eine $h_t$-sortierte, dann eine $h_{t-1}$-sortierte usw. und schließlich eine 1-sortierte, also eine sortierte Folge hergestellt.

**Beispiel:** Betrachten wir das Feld $a$ mit den sieben Schlüsseln 15, 2, 43, 17, 4, 8, 47 und die Folge 5, 3, 1 von Inkrementen. Wir betrachten zunächst die Elemente im Abstand 5 voneinander.

$$\underline{15} \quad \underline{2} \quad 43 \quad 17 \quad 4 \quad \underline{8} \quad \underline{47}$$

Um daraus mittels Sortieren durch Einfügen eine 5-sortierte Folge zu machen, wird das Element mit Schlüssel 15 mit dem Element mit Schlüssel 8 vertauscht und somit um fünf Positionen nach rechts verschoben. Außerdem wird 2 mit 47 verglichen, aber nicht vertauscht. Wir erhalten:

$$8 \quad 2 \quad 43 \quad 17 \quad 4 \quad 15 \quad 47$$

Jetzt betrachten wir alle Folgen von Elementen mit Abstand 3 und erhalten nach Sortieren:

$$8 \quad 2 \quad 15 \quad 17 \quad 4 \quad 43 \quad 47$$

Diese 3-sortierte Folge wird jetzt – wie beim Sortieren durch Einfügen – endgültig 1-sortiert. Dazu müssen jetzt nur noch insgesamt vier Verschiebungen um je eine Position nach rechts durchgeführt werden. Insgesamt wurden sechs Bewegungen (ein 5-er Sprung, ein 3-er Sprung und vier 1-er Sprünge) ausgeführt. Sortieren der Ausgangsfolge mit Einfügesort erfordert dagegen acht Bewegungen.

**procedure** *Shellsort* (**var** $a$ : *sequence*);

```
var
  i, j, k : integer;
  t : item; {Hilfsspeicher}
  continue : boolean; {für Schleifenabbruch}
begin
  for each h {einer endlichen, abnehmenden, mit 1 endenden
              Folge von Inkrementen} do
    {stelle h-sortierte Folge her}
    for i := h + 1 to N do
      begin
        j := i;
        t := a[i];
        k := t.key;
        continue := true;
        while (a[j − h].key > k) and continue do
          begin
            {h-Sprung nach rechts}
            a[j] := a[j − h];
            j := j − h;
            continue := (j > h)
          end;
        a[j] := t
      end
end
```

Die wichtigste Frage im Zusammenhang mit dem oben angegebenen Verfahren Shellsort ist, welche Folge von abnehmenden Inkrementen man wählen soll um die Gesamtzahl der Bewegungen möglichst gering zu halten. Auf diese Frage hat man inzwischen eine ganze Reihe überraschender, aber insgesamt doch nur unvollständiger Antworten erhalten. Beispielsweise kann man zeigen, dass die Laufzeit des Verfahrens $O(N\log^2 N)$ ist, wenn als Inkremente alle Zahlen der Form $2^p3^q$ gewählt werden, die kleiner als $N$ sind (vgl. [100]). Ein weiteres bemerkenswertes Resultat ist, dass das Herstellen einer $h$-sortierten Folge aus einer bereits $k$-sortierten Folge (wie im Verfahren Shellsort für abnehmende Inkremente $k$ und $h$) die $k$-Sortiertheit der Folge nicht zerstört.

### 2.1.4 Bubblesort

**Methode:** Lässt man als Bewegungen nur das wiederholte Vertauschen benachbarter Datensätze zu, so kann eine nach aufsteigenden Schlüsseln sortierte Folge von Datensätzen offensichtlich wie folgt hergestellt werden. Man durchläuft die Liste $a[1],\ldots,a[N]$ der Datensätze und betrachtet dabei je zwei benachbarte Elemente $a[i]$ und $a[i+1]$, $1 \le i < N$. Ist $a[i].key > a[i+1].key$, so vertauscht man $a[i]$ und $a[i+1]$. Nach diesem ersten Durchlauf ist das größte Element an seinem richtigen Platz am rechten Ende angelangt. Dann geht man die Folge erneut durch und vertauscht, falls nötig, wiederum je zwei benachbarte Elemente. Dieses Durchlaufen wird solange wiederholt,

bis keine Vertauschungen mehr aufgetreten sind; d. h. alle Paare benachbarter Sätze stehen in der richtigen Reihenfolge. Damit ist das Feld $a$ nach aufsteigenden Schlüsseln sortiert. Größere Elemente haben also die Tendenz, wie Luftblasen im Wasser langsam nach oben aufzusteigen. Diese Analogie hat dem Verfahren den Namen Bubblesort eingebracht.

**Beispiel:** Betrachten wir wieder das Feld $a$ mit den sieben Schlüsseln 15, 2, 43, 17, 4, 8, 47. Beim ersten Durchlauf werden folgende Vertauschungen benachbarter Elemente durchgeführt.

$$
\begin{array}{cccccccc}
15, & 2 & & & & & & \\
2, & 15, & 43, & 17 & & & & \\
 & & 17, & 43, & 4 & & & \\
 & & & 4, & 43, & 8 & & \\
 & & & & 8, & 43, & 47 &
\end{array}
$$

Nach dem ersten Durchlauf ist die Reihenfolge der Schlüssel des Feldes $a$ also

$$2,\ 15,\ 17,\ 4,\ 8,\ 43,\ 47.$$

Der zweite Durchlauf liefert die Reihenfolge

$$2,\ 15,\ 4,\ 8,\ 17,\ 43,\ 47.$$

Der dritte Durchlauf liefert schließlich

$$2,\ 4,\ 8,\ 15,\ 17,\ 43,\ 47,$$

also die aufsteigend sortierte Folge von Sätzen. Beim Durchlaufen dieser Folge müssen keine benachbarten Elemente mehr vertauscht werden. Das zeigt den erfolgreichen Abschluss des Sortierverfahrens. Man erhält damit das folgende nahe liegende Programmgerüst des Verfahrens:

```
procedure bubblesort (var a : sequence);
  var
    i : integer;
  begin
    repeat
      for i := 1 to (N − 1) do
        if a[i].key > a[i + 1].key
          then {vertausche a[i] und a[i + 1]}
    until {keine Vertauschung mehr aufgetreten}
  end
```

Bei Verwenden einer booleschen Variablen für den Test, ob Vertauschungen auftraten, kann das Verfahren wie folgt als Pascal-Prozedur geschrieben werden:

```
procedure bubblesort (var a : sequence);
  var
```

```
    i : integer;
    nichtvertauscht : boolean;
    t : item; {Hilfsspeicher}
begin
  repeat
    nichtvertauscht := true;
    for i := 1 to (N − 1) do
{∗}   if a[i].key > a[i + 1].key
        then
          begin
{∗∗}        t := a[i];
{∗∗}        a[i] := a[i + 1];
{∗∗}        a[i + 1] := t;
            nichtvertauscht := false
          end
  until nichtvertauscht
end
```

An dieser Stelle wollen wir noch auf eine kleine Effizienzverbesserung mithilfe eines Programmiertricks hinweisen. Sind alle Schlüssel verschieden, so kann man die Prüfung, ob bei einem Durchlauf noch eine Vertauschung auftrat, ohne eine boolesche Variable wie folgt testen. Man besetzt zu Beginn der **repeat**-Schleife die für die Vertauschung vorgesehene Hilfsspeichervariable $t$ mit dem Wert von $a[1]$. Tritt beim Durchlaufen wenigstens eine Vertauschung zweier Elemente $a[i]$ und $a[i+1]$ mit $i > 1$ auf, hat $t$ am Ende des Durchlaufs nicht mehr denselben Wert wie zu Beginn. Der Wert von $t$ kann am Ende des Durchlaufs höchstens dann noch den ursprünglichen Wert $a[1]$ haben, wenn entweder keine Vertauschung aufgetreten ist oder die einzige beim Durchlauf vorgenommene Vertauschung die der Elemente $a[1]$ und $a[2]$ war. In beiden Fällen ist das Feld am Ende des Durchlaufs sortiert.

Die eben skizzierte Möglichkeit zur Implementation des Verfahrens Bubblesort ist in ganz seltenen Fällen besser als die von uns angegebene, da unter Umständen ein „unnötiges“ Durchlaufen eines bereits nach aufsteigenden Schlüsseln sortierten Feldes vermieden wird.

Wir haben stets das ganze Feld durchlaufen, obwohl das natürlich nicht nötig ist. Denn nach dem $i$-ten Durchlauf befinden sich die $i$ größten Elemente bereits am rechten Ende. Man erhält also eine Effizienzverbesserung auch dadurch, dass man im $i$-ten Durchlauf nur die Elemente an den Positionen $1,\ldots,(N-i)+1$ inspiziert. Wir überlassen es dem Leser, sich zu überlegen, wie man das implementieren kann.

**Analyse:** Die Abschätzung der im günstigsten und schlechtesten Fall ausgeführten Anzahlen von Schlüsselvergleichen (in Programmzeile $\{*\}$) und Bewegungen von Datensätzen (in Programmzeilen $\{**\}$) ist einfach. Ist das Feld $a$ bereits nach aufsteigenden Schlüsseln sortiert, so wird die **for**-Schleife des oben angegebenen Programms genau einmal durchlaufen und dabei keine Vertauschung vorgenommen. Also ist

$$C_{min}(N) = N-1, \qquad M_{min}(N) = 0.$$

Der ungünstigste Fall für das Verfahren Bubblesort liegt vor, wenn das Feld $a$ anfangs nach absteigenden Schlüsseln sortiert ist. Dann rückt das Element mit minimalem Schlüssel bei jedem Durchlauf der **repeat**-Schleife um eine Position nach vorn. Es sind dann $N$ Durchläufe nötig, bis es ganz vorn angelangt ist und festgestellt wird, dass keine Vertauschung mehr aufgetreten ist (wendet man den erwähnten Programmiertrick an, so sind es nur $N-1$ Durchläufe). Es ist nicht schwer zu sehen, dass in diesem Fall beim $i$-ten Durchlauf, $1 \leq i < N$, $(N-i)$ Vertauschungen benachbarter Elemente, also $3(N-i)$ Bewegungen, und natürlich jedes Mal $N-1$ Schlüsselvergleiche ausgeführt werden. Damit ist:

$$\begin{aligned} C_{max} &= N(N-1) = \Theta(N^2) \\ M_{max} &= \sum_{i=1}^{N-1} 3(N-i) = \Theta(N^2) \end{aligned}$$

Man kann zeigen, dass auch

$$C_{mit}(N) = M_{mit}(N) = \Theta(N^2)$$

gilt (vgl. [100]).

Wir verzichten hier auf diesen Nachweis, denn Bubblesort ist ein zwar durchaus populäres, aber im Grunde schlechtes elementares Sortierverfahren. Nur für den Fall, dass ein bereits nahezu vollständig sortiertes Feld (für das die Inversionszahl der Schlüsselfolge klein ist) vorliegt, werden wenige Vergleichsoperationen und Bewegungen von Datensätzen ausgeführt. Das Verfahren ist stark asymmetrisch bezüglich der Durchlaufrichtung. Ist z. B. die Ausgangsfolge schon „fast sortiert“, d. h. gilt für $k_1, \ldots, k_N$

$$k_i \leq k_{i+1},\ 1 \leq i < N-1,$$

und ist $k_N$ das minimale Element, so sind $N-1$ Durchläufe nötig um es an den Anfang zu schaffen. Man hat versucht diese Schwäche dadurch zu beheben, dass man das Feld $a$ abwechselnd von links nach rechts und umgekehrt durchläuft. Diese (geringfügig bessere) Variante ist als *Shakersort* bekannt. Außer einem schönen Namen hat das Verfahren aber keine Vorzüge, wenn man es etwa mit dem Verfahren Sortieren durch Einfügen vergleicht.

## 2.2 Quicksort

Wir stellen in diesem Abschnitt ein Sortierverfahren vor, das 1962 von C. A. R. Hoare [89] veröffentlicht wurde und den Namen Quicksort erhielt, weil es erfahrungsgemäß eines der schnellsten, wenn nicht *das* schnellste interne Sortierverfahren ist. Das Verfahren folgt der Divide-and-conquer-Strategie zur Lösung des Sortierproblems. Es benötigt zwar, wie die elementaren Sortierverfahren, $\Omega(N^2)$ viele Vergleichsoperationen zwischen Schlüsseln im schlechtesten Fall, im Mittel werden jedoch nur $O(N \log N)$ viele Vergleichsoperationen ausgeführt. Quicksort operiert auf den Elementen eines Feldes $a$

von Datensätzen mit Schlüsseln, die wir ohne Einschränkung als ganzzahlig annehmen. Es ist ein so genanntes In-situ-Sortierverfahren. Das bedeutet, dass zur (Zwischen-) Speicherung für die Datensätze kein zusätzlicher Speicher benötigt wird, außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen. Nur für die Verwaltung der Information über noch nicht vollständig abgearbeitete und durch rekursive Anwendung der Divide-and-conquer-Strategie generierte Teilprobleme wird zusätzlicher Speicherplatz benötigt. Quicksort kann auf viele verschiedene Arten implementiert werden. Wir geben im Abschnitt 2.2.1 eine nahe liegende Version an und analysieren das Verhalten im schlechtesten Fall, im besten Fall und im Mittel. Im Abschnitt 2.2.2 besprechen wir einige Varianten des Verfahrens, die unter bestimmten Voraussetzungen ein besseres Verhalten liefern. Als Beispiel behandeln wir insbesondere den Fall, dass das zu sortierende Feld viele Sätze mit gleichen Schlüsseln hat – ein in der Praxis durchaus nicht seltener Fall.

Quicksort ist sehr empfindlich gegen minimale Programmänderungen. Jede Version einer Implementation muss sorgfältig daraufhin überprüft werden, ob sie auch wirklich in allen Fällen das korrekte Ergebnis liefert.

## 2.2.1 Quicksort: Sortieren durch rekursives Teilen

**Methode:** Um eine Folge $F = k_1, \ldots, k_N$ von $N$ Schlüsseln nach aufsteigenden Werten zu sortieren wählen wir ein beliebiges Element $k \in \{k_1, \ldots, k_N\}$ und benutzen es als Angelpunkt, genannt *Pivotelement*, für eine Aufteilung der Folge ohne $k$ in zwei Teilfolgen $F_1$ und $F_2$. $F_1$ besteht nur aus Elementen von $F$, die kleiner oder gleich $k$ sind, $F_2$ nur aus Elementen von $F$, die größer oder gleich $k$ sind. Ist $F_1$ eine Folge mit $i-1$ Elementen und $F_2$ eine Folge mit $N-i$ Elementen, so ist $i$ die endgültige Position des Pivotelements $k$. Also kann man das Sortierproblem dadurch lösen, dass man $F_1$ und $F_2$ rekursiv auf dieselbe Weise sortiert und die Ergebnisse in offensichtlicher Weise zusammensetzt. Zuerst kommt die durch Sortieren von $F_1$ entstandene Folge, dann das Pivotelement $k$ (an Position $i$) und dann die durch Sortieren von $F_2$ entstandene Folge.

Lässt man alle Implementationsdetails zunächst weg, so kann die Struktur des Verfahrens auf einer hohen sprachlichen Ebene wie folgt beschrieben werden.

> **Algorithmus** *Quicksort* $(F : Folge)$;
> $\{$*sortiert die Folge F nach aufsteigenden Werten*$\}$
> *Falls F die leere Folge ist oder F nur aus einem einzigen Element besteht, bleibt F unverändert; sonst:*
>
> *Divide: Wähle ein Pivotelement k von F* $(z.B.$ *das letzte*$)$ *und teile F ohne k in Teilfolgen* $F_1$ *und* $F_2$ *bzgl. k:*
> $F_1$ *enthält nur Elemente von F ohne k, die* $\leq k$ *sind,*
> $F_2$ *enthält nur Elemente von F ohne k, die* $\geq k$ *sind;*
>
> *Conquer: Quicksort*$(F_1)$*; Quicksort*$(F_2)$*;*
> $\{$*nach Ausführung dieser beiden Aufrufe sind* $F_1$ *und* $F_2$ *sortiert*$\}$
>
> *Merge: Bilde die Ergebnisfolge F durch Hintereinanderhängen von* $F_1$*, k,* $F_2$ *in dieser Reihenfolge.*

Der für die Implementation des Verfahrens wesentliche Schritt ist der Aufteilungsschritt. Die Aufteilung bzgl. eines gewählten Pivotelementes soll in situ, d. h. am Ort, an dem die ursprünglichen Sätze abgelegt sind, ohne zusätzlichen, von der Anzahl der zu sortierenden Folgenelemente abhängigen Speicherbedarf erfolgen. Die als Ergebnis der Aufteilung entstehenden Teilfolgen $F_1$ und $F_2$ könnte man programmtechnisch als Arrays geringerer Länge zu realisieren versuchen. Das würde bedeuten, eine rekursive Prozedur *quicksort* zu schreiben mit einem Array variabler Länge als Ein- und Ausgabeparameter. Das ist in der Programmiersprache Pascal nicht möglich – und glücklicherweise auch nicht nötig. Wir schreiben eine Prozedur, die das als Ein- und Ausgabeparameter gegebene Feld $a$ der Datensätze verändert. Die zwischendurch durch Aufteilen entstandenen Teilfolgen werden durch ein Paar von Zeigern (Indizes) auf das Array realisiert. Diese Zeiger werden Eingabeparameter der Prozedur *quicksort*.

```
procedure quicksort (var a : sequence; l, r : integer);
{sortiert die Elemente a[l],...,a[r] des Feldes a
 nach aufsteigenden Schlüsseln}
```

Ein Aufruf der Prozedur $quicksort(a,1,N)$ sortiert also die gegebene Folge von Datensätzen.

Als Pivotelement $v$ wollen wir den Schlüssel des Elements $a[r]$ am rechten Ende des zu sortierenden Teilfeldes wählen. Eine In-situ-Aufteilung des Bereiches $a[l],\ldots,a[r]$ bzgl. $v$ kann man nun wie folgt erreichen. Man wandert mit einem Positionszeiger $i$ vom linken Ende des aufzuteilenden Bereiches nach rechts über alle Elemente hinweg, deren Schlüssel kleiner ist als $v$, bis man schließlich ein Element mit $a[i].key \geq v$ trifft. Symmetrisch dazu wandert man mit Zeiger $j$ vom rechten Ende des aufzuteilenden Bereiches nach links über alle Elemente hinweg, deren Schlüssel größer ist als $v$, bis man schließlich ein Element mit $a[j].key \leq v$ trifft. Dann vertauscht man $a[i]$ mit $a[j]$, wodurch beide bezüglich $v$ in der richtigen Teilfolge stehen.

Das wird solange wiederholt, bis die Teilfolge $a[l],\ldots,a[r]$ vollständig inspiziert ist. Das kann man daran feststellen, dass die Zeiger $i$ und $j$ übereinander hinweg gelaufen sind. Wenn dieser Fall eintritt, hat man zugleich auch die endgültige Position des Pivotelementes gefunden. Wir wollen jetzt dieses Vorgehen als Pascal-Prozedur realisieren. Dazu ist es bequem die sprachlichen Möglichkeiten von Pascal um ein allgemeineres Schleifenkonstrukt zu erweitern. Wir verwenden eine Schleife der Form

```
begin-loop
   <statement>
   if <condition>
      then exit-loop;
   <statement>
end-loop
```

mit offensichtlicher Bedeutung.

```
procedure quicksort (var a : sequence; l, r : integer);
var
  v, i, j : integer;
  t : item; {Hilfsspeicher}
begin
  if r > l
  then
    begin
      i := l − 1;
      j := r;
      v := a[r].key; {Pivotelement}
      begin-loop
{∗}     repeat i := i + 1 until a[i].key ≥ v;
{∗}     repeat j := j − 1 until a[j].key ≤ v;
        if i ≥ j
          then {i ist Pivotposition}
            exit-loop;
{∗∗}    t := a[i];
{∗∗}    a[i] := a[j];
{∗∗}    a[j] := t
      end-loop;
{∗∗}  t := a[i];
{∗∗}  a[i] := a[r];
{∗∗}  a[r] := t;
      quicksort(a, l, i − 1);
      quicksort(a, i + 1, r)
    end
end
```

Wir erläutern den Aufteilungsschritt am Beispiel eines Aufrufs von $quicksort(a, 4, 9)$ für den Fall, dass die Folge der Schlüssel im Bereich $a[4] \ldots a[9]$ die Schlüssel 5, 7, 3, 1, 6, 4 sind, vgl. Tabelle 2.1.

Da wir als Pivotelement $v$ das Element am rechten Ende des aufzuteilenden Bereichs gewählt haben, ist klar, dass es im Bereich, den der Zeiger $i$ beim Wandern nach rechts überstreicht, stets ein Element mit Schlüssel $\geq v$ gibt. Die erste der beiden **repeat**-Schleifen terminiert also sicher. Die zweite **repeat**-Schleife terminiert aber dann nicht korrekt, wenn das Pivotelement der minimale Schlüssel unter allen Schlüsseln im Bereich $a[l] \ldots a[r]$ ist. Dann gibt es nämlich kein $j \in \{r-1, r-2, \ldots, l\}$ mit $a[j].key \leq v$. Die zweite **repeat**-Schleife terminiert sicher dann, wenn an Position $l-1$ ein Element steht, für das gilt $a[l-1].key \leq a[j].key$ für alle $j$ mit $l \leq j \leq r$. Das kann man für den ersten Aufruf $quicksort(a, 1, N)$ sichern durch Abspeichern eines Stoppers an Position 0 mit $a[0].key \leq \min_i\{a[i].key\}$. Bei allen rekursiven Aufrufen ist die entsprechende Bedingung von selbst gesichert. Das zeigt folgende Überlegung. Unter der Annahme, dass es vor einem Aufruf von $quicksort(a, l, r)$ ein Element an Position $l-1$ gibt mit $a[l-1].key \leq a[j].key$ für alle $j$ mit $l \leq j \leq r$, gilt die entsprechende Bedingung auch, bevor die rekursiven Aufrufe $quicksort(a, l, i-1)$ und $quicksort(a, i+1, r)$ ausgeführt werden. Das ist trivial für den erstgenannten Aufruf. Ferner hat die vorangehende Auf-

| Array-Position | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ⋯ |
|---|---|---|---|---|---|---|---|---|---|
| Schlüssel | ⋯ | 5 | 7 | 3 | 1 | 6 | $\boxed{4}$ | ⋯ | 4 ist Pivot-Element |
| | | $\uparrow_i$ | | | | | $\uparrow_j$ | | |
| | | | $\uparrow_i$ | | $\uparrow_j$ | | | | 1. Halt der Zeiger $i$, $j$ |
| | ⋯ | 1 | 7 | 3 | 5 | 6 | $\boxed{4}$ | ⋯ | |
| | | | | $\uparrow_i$ | $\uparrow_j$ | | | | 2. Halt der Zeiger $i$, $j$ |
| | ⋯ | 1 | 3 | 7 | 5 | 6 | $\boxed{4}$ | ⋯ | |
| | | | $\uparrow_j$ | $\uparrow_i$ | | | | | letzter Halt der Zeiger $i$, $j$ |
| | ⋯ | 1 | 3 | $\boxed{4}$ | 5 | 6 | 7 | ⋯ | |
| | | | $\uparrow_j$ | $\uparrow_i$ | | | | | |

**Tabelle 2.1**

teilung bewirkt, dass an Position $i$ ein Element steht mit $a[i].key \leq a[j].key$ für alle $j$ mit $i+1 \leq j \leq r$.

Wir haben die Abbruchbedingungen für die **repeat**-Schleifen übrigens so gewählt, dass das Verfahren auch auf Felder anwendbar ist, in denen dieselben Schlüssel mehrfach auftreten. Es werden in diesem Fall allerdings unnötige Vertauschungen vorgenommen und die Elemente mit gleichem Schlüssel können ihre relative Position ändern. Wir zeigen im Abschnitt 2.2.2 eine Möglichkeit das zu vermeiden.

**Analyse:** Wir schätzen zunächst die im *schlechtesten Fall* bei einem Aufruf von $quicksort(a,1,N)$ auszuführende Anzahl von Schlüsselvergleichen (in den Programmzeilen $\{*\}$) sowie die Anzahl der Bewegungen (in Programmzeilen $\{**\}$) ab. Zur Aufteilung eines Feldes der Länge $N$ werden die Schlüssel aller Elemente im aufzuteilenden Bereich mit dem Pivotelement verglichen. In der Regel werden zwei Schlüssel je zweimal mit dem Pivotelement verglichen. Es sind immer $N+1$ Vergleiche insgesamt, wenn das Pivotelement in den $N-1$ Restelementen nicht vorkommt, sonst $N$ Vergleiche. Im ungünstigsten Fall wechseln dabei alle Elemente je einmal ihren Platz. Die im ungünstigsten Fall auszuführende Anzahl von Schlüsselvergleichen und Bewegungen hängt damit stark von der Anzahl der Aufteilungsschritte und damit von der Zahl der initiierten rekursiven Aufrufe ab. Ist das Pivotelement das Element mit kleinstem oder größtem Schlüssel, ist eine der beiden durch Aufteilung entstehenden Folgen jeweils leer und die andere hat jeweils ein Element (nämlich das Pivotelement) weniger als die Ausgangsfolge. Dieser Fall tritt z. B. für eine bereits aufsteigend sortierte Folge von Schlüsseln ein. Die in diesem Fall initiierte Folge von rekursiven Aufrufen kann man durch den Baum in Abbildung 2.1 veranschaulichen.

Damit ist klar, dass zum Sortieren von $N$ Elementen mit Quicksort für die maximale Anzahl $C_{max}(N)$ von auszuführenden Schlüsselvergleichen gilt

$$C_{max}(N) \geq \sum_{i=2}^{N} (i+1) = \Omega(N^2).$$

Quicksort benötigt im schlechtesten Fall also quadratische Schrittzahl.

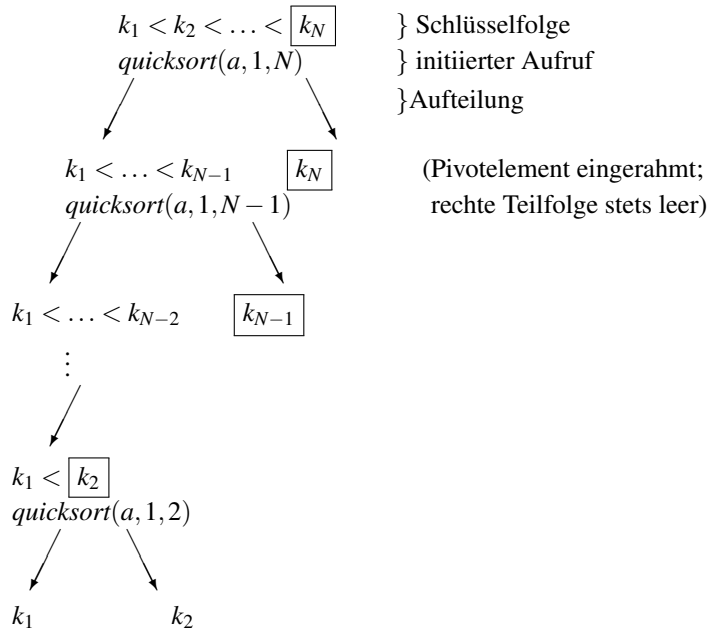



**Abbildung 2.1**

Im günstigsten Fall haben die durch Aufteilung entstehenden Teilfolgen stets etwa gleiche Länge. Dann hat der Baum der initiierten rekursiven Aufrufe die minimale Höhe (ungefähr $\log N$) wie im Beispiel von Abbildung 2.2 mit 15 Schlüsseln.

Zur Aufteilung aller Folgen auf einem Niveau werden $\Theta(N)$ Schlüsselvergleiche durchgeführt. Da der Aufrufbaum im günstigsten Fall nur die Höhe $\log N$ hat, folgt unmittelbar

$$C_{min}(N) = \Theta(N \log N).$$

Es ist nicht schwer zu sehen, dass die Gesamtlaufzeit von Quicksort im günstigsten Fall durch $\Theta(N \log N)$ abgeschätzt werden kann.

Wir überlegen uns noch, dass bei einem Aufruf von $quicksort(a,1,N)$ stets höchstens $O(N \log N)$ Bewegungen vorkommen.

Bei jeder Vertauschung zweier Elemente (Zeilen $\{**\}$) innerhalb eines Aufteilungsschritts ist das eine kleiner als das Pivotelement, das andere größer; nur am Schluss des Aufteilungsschritts ist das Pivotelement selbst beteiligt. Also ist die Anzahl der Vertauschungen bei einem Aufteilungsschritt höchstens so groß wie die Anzahl der Elemente in der kürzeren der beiden entstehenden Teilfolgen. Belasten wir nun die Kosten für das Bewegen von Elementen bei diesem Vertauschen demjenigen beteiligten Element, das in der kürzeren Teilfolge landet, so wird jedes Element bei einem Aufteilungsschritt höchstens mit konstanten Kosten belastet (für die Vertauschung, an der es beteiligt war). Verfolgen wir nun ein einzelnes Element über den ganzen Sortierprozess hinweg und beobachten wir dabei die Längen der Teilfolgen, in denen sich

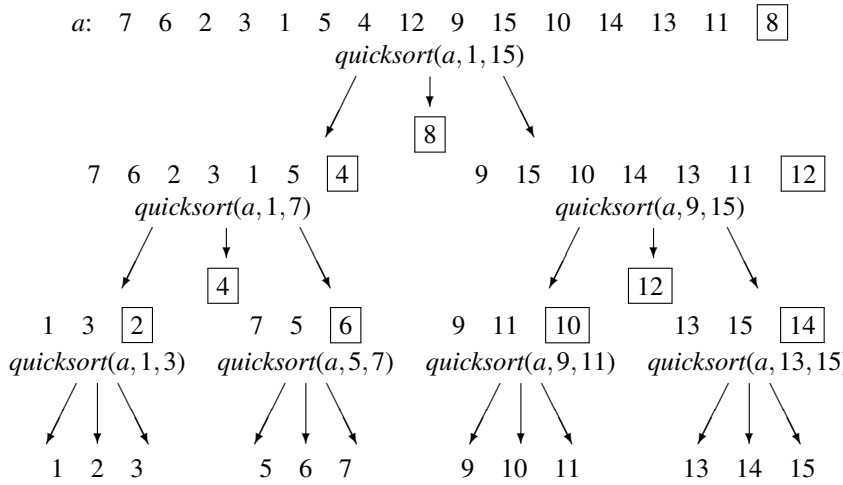



**Abbildung 2.2**

dieses Element befindet. Wann immer dieses Element mit Bewegungs-Kosten belastet wird, hat sich die Länge der Teilfolge auf höchstens die Hälfte reduziert. Das kann aber höchstens $\log N$ Mal passieren, bevor die Teilfolge nur noch ein Element enthält. Da diese Überlegung für jedes beliebige der $N$ Elemente gilt, ergibt sich als obere Schranke für die Anzahl der Bewegungen von Schlüsseln bei Quicksort $O(N \log N)$. Bezüglich der Vertauschungen, also der Anzahl der ausgeführten Bewegungen von Datensätzen, ist das Aufspalten des zu sortierenden Bereichs in der Mitte folglich der ungünstigste Fall. Der Fall, dass man im Aufteilungsschritt immer wieder nur einen einzigen Datensatz abspalten kann, ist dagegen der günstigste: Es wird nur ein Datensatz bewegt. Die Verhältnisse sind also gerade umgekehrt wie bei der Anzahl der ausgeführten Schlüsselvergleiche.

Wir wollen jetzt zeigen, dass die mittlere Laufzeit von Quicksort nicht viel schlechter ist als die Laufzeit im günstigsten Fall. Um das zu zeigen, gehen wir von folgenden Annahmen aus. Erstens nehmen wir an, dass alle $N$ Schlüssel paarweise verschieden voneinander sind. Wir können daher für Quicksort ohne Einschränkung voraussetzen, dass die Schlüssel die Zahlen $1,\ldots,N$ sind. Zweitens betrachten wir jede der $N!$ möglichen Anordnungen von $N$ Schlüsseln als gleich wahrscheinlich.

Wird Quicksort für eine Folge $k_1,\ldots,k_N$ von Schlüsseln aufgerufen, so folgt aus den Annahmen, dass jede Zahl $k$, $1 \leq k \leq N$, mit gleicher Wahrscheinlichkeit $1/N$ an Position $N$ auftritt und damit als Pivotelement gewählt wird. Wird $k$ Pivotelement, so werden durch Aufteilung zwei Folgen mit Längen $k-1$ und $N-k$ erzeugt, für die *quicksort* rekursiv aufgerufen wird. Man kann nun zeigen, dass die durch Aufteilung entstehenden Teilfolgen wieder „zufällig“ sind, wenn die Ausgangsfolge „zufällig“ war. Durch Aufteilung sämtlicher Folgen $k_1,\ldots,k_N$ mit $k_N = k$ erhält man wieder sämtliche Folgen von $k-1$ und $N-k$ Elementen. Das erlaubt es unmittelbar eine Rekursionsformel für die mittlere Laufzeit $T(N)$ des Verfahrens Quicksort zum Sortieren von $N$ Schlüsseln aufzustellen. Offensichtlich ist $T(1) = a$, für eine Konstante $a$. Falls $N \geq 2$ ist, gilt mit

einer Konstanten $b$

$$T(N) \leq \frac{1}{N} \cdot \sum_{k=1}^{N} (T(k-1) + T(N-k)) + bN.$$

In dieser Formel gibt der Term $bN$ den Aufteilungsaufwand für eine Folge der Länge $N$ an. Es folgt für $N \geq 2$, da $T(0) = 0$ ist,

$$T(N) \leq \frac{2}{N} \cdot \sum_{k=1}^{N-1} T(k) + bN.$$

Wir zeigen per Induktion, dass hieraus

$$T(N) \leq c \cdot N \log N$$

für $N \geq 2$ mit einer hinreichend groß gewählten Konstanten c folgt (dabei nehmen wir an, dass $N$ gerade ist; der Fall, dass $N$ ungerade ist, lässt sich analog behandeln).

Der Induktionsanfang ist klar. Sei nun $N \geq 3$ und setzen wir für alle $i < N$ voraus, dass bereits $T(i) \leq c \cdot i \log i$ gilt. Dann folgt:

$$\begin{aligned}
T(N) &\leq \frac{2}{N}\left[\sum_{k=1}^{N-1} T(k)\right] + bN \\
&\leq \frac{2c}{N}\left[\sum_{k=1}^{N-1} k \cdot \log k\right] + bN \\
&= \frac{2c}{N}\left[\sum_{k=1}^{\frac{N}{2}} k \cdot \underbrace{\log k}_{\leq \log N - 1} + \sum_{k=1}^{\frac{N}{2}-1} \left(\frac{N}{2}+k\right) \underbrace{\log\left(\frac{N}{2}+k\right)}_{\leq \log N}\right] + bN \\
&\leq \frac{2c}{N}\left[\frac{N}{4}\left(\frac{N}{2}+1\right)\log N - \frac{N^2}{8} - \frac{N}{4} + \left(\frac{3N^2}{8} - \frac{3N}{4}\right)\log N\right] + bN \\
&= \frac{2c}{N}\left[\left(\frac{N^2}{2} - \frac{N}{2}\right)\log N - \frac{N^2}{8} - \frac{N}{4}\right] + bN \\
&= c \cdot N \log N - \underbrace{c \cdot \log N}_{\geq 0} - \frac{cN}{4} - \frac{c}{2} + bN \\
&\leq c \cdot N \log N - \frac{cN}{4} - \frac{c}{2} + bN
\end{aligned}$$

Haben wir jetzt $c \geq 4b$ gewählt, so folgt unmittelbar

$$T(N) \leq c \cdot N \log N.$$

Damit ist bewiesen, dass Quicksort im Mittel $O(N \log N)$ Zeit benötigt. Wir haben die gesamte mittlere Laufzeit von Quicksort abgeschätzt. Eine entsprechende Rekursionsgleichung gilt natürlich auch für die mittlere Anzahl von Schlüsselvergleichen, die damit ebenfalls im Mittel $O(N \log N)$ ist.

Quicksort benötigt außer einer einzigen Hilfsspeicherstelle beim Aufteilen eines Feldes keinen zusätzlichen Speicher zur Zwischenspeicherung von Datensätzen. Wie bei jeder rekursiven Prozedur muss aber Buch geführt werden über die begonnenen, aber noch nicht abgeschlossenen rekursiven Aufrufe der Prozedur *quicksort*. Das können bis zu $\Omega(N)$ viele Aufrufe sein. Eine Möglichkeit, den zusätzlich benötigten Speicherplatz in der Größenordnung von $O(\log N)$ zu halten, besteht darin, jeweils das kleinere der beiden durch Aufteilung erhaltenen Teilprobleme zuerst (rekursiv) zu lösen. Das größere Teilproblem kann dann nicht auch rekursiv gelöst werden, weil sonst die Schachtelungstiefe der rekursiven Aufrufe weiterhin linear in der Anzahl der Folgenelemente sein könnte, wie etwa im Falle einer vorsortierten Folge. Man behilft sich, indem man die sich ergebenden größeren Teilprobleme durch eine Iteration löst. Damit ergibt sich folgender Block der Prozedur *quicksort*.

```
begin {quicksort mit logarithmisch beschränkter Rekursionstiefe}
    while r > l do
      begin
        {wähle Pivot-Element und teile Folge auf wie bisher}
        {statt zweier rekursiver Aufrufe verfahre wie folgt:}
        if (i-1-l) ≤ (r-i-1)
          then
            begin
              {rekursiver Aufruf für a[l]...a[i-1]}
              quicksort(a,l,i-1);
              {Iteration für a[i+1]...a[r]}
              l := i+1
            end
          else
            begin
              {rekursiver Aufruf für a[i+1]...a[r]}
              quicksort(a,i+1,r);
              {Iteration für a[l]...a[i-1]}
              r := i-1
            end
      end
  end {Quicksort}
```

Natürlich kann man Quicksort auch gänzlich iterativ programmieren, indem man sich die Indizes für das linke und das rechte Ende der noch zu sortierenden Teilfolgen merkt (z. B. mithilfe eines Stapels, vgl. Kapitel 1). Sortiert man die jeweils kleinere Teilfolge zuerst, so muss man sich nie mehr als $O(\log N)$ Indizes merken.

Nach einem Vorschlag von B. Ďurian [45] kann man Quicksort auch mit nur konstantem zusätzlichem Speicherplatz realisieren, ein wenig zulasten der Laufzeit. Hier merkt man sich die noch zu sortierenden Teilfolgen nicht explizit, sondern sucht sie in der Gesamtfolge auf. Nach einer Aufteilung wird zuerst die linke, dann die rechte Teilfolge sortiert. Betrachten wir einen Ausschnitt aus dem Ablauf des Sortierprozesses, wie ihn Abbildung 2.3 zeigt.

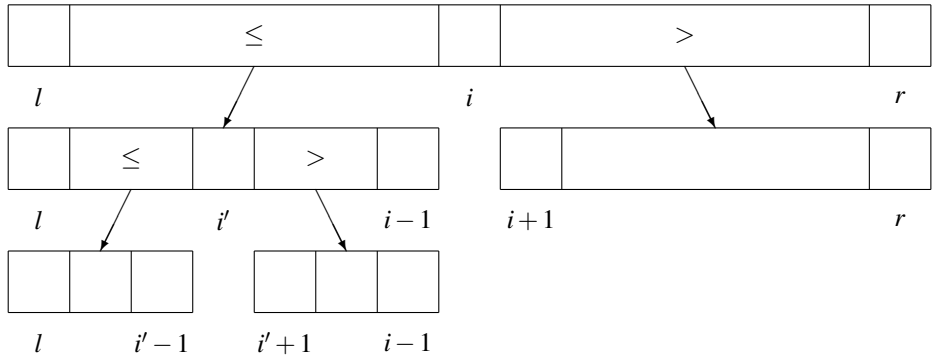



**Abbildung 2.3**

Von den gezeigten Teilfolgen wird also zuerst $a[l]\ldots a[i'-1]$ sortiert, dann $a[i'+1]\ldots a[i-1]$ und schließlich $a[i+1]\ldots a[r]$. Das Problem ist nun, dass man zum Sortieren der Teilfolge $a[i'+1]\ldots a[i-1]$ die rechte Grenze, also den Index $i-1$, kennen muss. Bisher haben wir uns dies implizit in der Rekursion oder explizit im Stapel gemerkt. Jetzt nutzen wir die Kenntnis aus, dass alle Schlüssel in der Teilfolge $a[i'+1]\ldots a[i-1]$ höchstens so groß wie das Pivotelement $a[i]$ sein können; alle Schlüssel in $a[i+1]\ldots a[r]$ müssen größer sein. Wir können also, ausgehend von $a[i']$, den Index $i-1$ finden, wenn wir $a[i].key$ kennen, etwa so:

$\{$*Sei* $v := a[i].key$*, der Schlüssel des Pivotelements*$\}$
  $m := i'$;
  **while** $a[m].key \leq v$ **do** $m := m+1$;
  $\{$*jetzt ist* $m = i+1\}$
  $m := m-2$;
  $\{$*jetzt ist* $m = i-1$*, der gewünschte Index*$\}$

Nun muss man natürlich noch $a[i].key$ kennen ohne den Index $i$ gespeichert zu haben. Das ist aber leicht möglich, wenn wir vor dem Sortieren der Elemente $a[l]\ldots a[i'-1]$ das Element $a[i]$ mit dem Element $a[i'+1]$ tauschen. Dann ergibt sich

$$v := a[i'+1].key$$

vor Beginn des gerade angegebenen Programmstücks; das Ausfüllen des Rests des Blockes der Prozedur *quicksort* überlassen wir dem interessierten Leser.

Das asymptotische Laufzeitverhalten von Quicksort ändert sich durch die zusätzlichen Vergleichs- und Bewegeoperationen nicht, da ja bereits der Aufteilungsschritt lineare Zeit kostet. Verwendet man statt sequenzieller Suche binäre Suche nach Position $i-1$, so ergibt sich eine nur wenig höhere Laufzeit als bei rekursivem Quicksort.

### 2.2.2 Quicksort-Varianten

Das im vorigen Abschnitt angegebene Verfahren Quicksort benötigt für bereits sortierte oder fast sortierte Eingabefolgen quadratische Schrittzahl. Der Grund dafür ist, dass in diesen Fällen die Wahl des Pivotelementes am rechten Ende des aufzuteilenden Bereichs keine gute Aufteilung des Feldes in zwei nahezu gleich große Teilfelder liefert. Es gibt mehrere Strategien für eine bessere Wahl des Pivotelementes. Die bekanntesten sind die *3-Median-* und die *Zufalls-Strategie*.

Im Falle der *3-Median-Strategie* wird als Pivotelement der Median (d. h. das mittlere) von drei Elementen im aufzuteilenden Bereich gewählt. Wählt man die drei Elemente vom linken und rechten Ende und aus der Mitte, so besteht eine gute Chance dafür, dass das mittlere dieser drei Elemente eine Aufteilung in annähernd gleiche Teile liefert. Um das mittlere von drei Elementen $a,b,c$ zu bestimmen, die nicht paarweise verschieden sein müssen, kann man wie folgt vorgehen.

```
if a > b then vertausche (a,b);
{a = min(a,b)}
if a > c then vertausche (a,c);
{a = min(a,b,c)}
if b > c then vertausche (b,c);
{a,b,c sind jetzt aufsteigend sortiert; also ist b das mittlere
 der drei Elemente a,b,c}
```

Setzt man das Element mit dem mittleren der drei Schlüssel $a = a[l].key$, $b = a[r].key$ und $c = a[m].key$ mit $m = (l + r)$ **div** 2 vor Beginn der Aufteilung des Bereichs $a[l] \ldots a[r]$ an das rechte Ende des aufzuteilenden Bereichs, kann die Aufteilung wie bisher erfolgen. Insgesamt erhalten wir folgende Prozedur:

```
procedure median_of_three_quicksort (var a : sequence; l, r : integer);
var
  v, m, i, j : integer;
  t : item; {Hilfsspeicher}
begin
  if r > l
    then
      begin
        m := (r+l) div 2;
        if a[l].key > a[r].key
          then
            begin
              t := a[l];
              a[l] := a[r];
              a[r] := t
            end;
        if a[l].key > a[m].key
          then
```

```
                begin
                  t := a[l];
                  a[l] := a[m];
                  a[m] := t
                end;
            if a[r].key > a[m].key
              then
                begin
                  t := a[r];
                  a[r] := a[m];
                  a[m] := t
                end;
              {jetzt steht Median von a[l], a[m] und a[r] an Position r;
               weiter wie bisher ...}
          end
      end
```

Im Falle der *Zufalls-Strategie* wählt man das Pivotelement zufällig unter den Schlüsseln im aufzuteilenden Bereich $a[l]\ldots a[r]$. Statt einfach $v := a[r].key$ zu setzen, wählt man zunächst $k$ zufällig und gleich verteilt aus dem Bereich der Indizes $l,\ldots,r$ und vertauscht $a[k]$ mit $a[r]$, bevor mit der Aufteilung des Bereichs $a[l]\ldots a[r]$ begonnen wird. Der Effekt dieser Änderung von Quicksort ist drastisch. Es gibt keine „schlechten“ Eingabefolgen mehr! Das auf diese Weise *randomisierte* (zufällig gemachte) *Quicksort* behandelt alle Eingabefolgen (annähernd) gleich. Natürlich kann man auch so nicht vermeiden, dass ein schlechtester Fall auftritt, in dem das Verfahren quadratische Schrittzahl benötigt. Man kann aber leicht zeigen (vgl. z. B. [135]), dass der Erwartungswert für die zum Sortieren einer beliebigen, aber festen Eingabefolge mit randomisiertem Quicksort erforderliche Anzahl von Schlüsselvergleichen gleich $O(N\log N)$ ist. In Abschnitt 11.1.1 werden wir noch mal ausführlicher auf randomisiertes Quicksort eingehen.

Ob die Implementation und Verwendung von randomisiertem Quicksort zweckmäßig ist, hängt vom jeweiligen Anwendungsfall ab. Im Falle stark vorsortierter Eingabefolgen kann es unter Umständen ausreichen die Eingabefolgen zunächst einmal „zufällig“ zu permutieren und darauf das normale Quicksort-Verfahren anzuwenden.

Die von uns im Abschnitt 2.2.1 angegebene Version des Verfahrens Quicksort lässt gleiche Schlüssel in der Eingabefolge zu. Nicht selten treten in Anwendungen Folgen mit vielen Wiederholungen auf. Man denke etwa an eine Datei mit offen stehenden Kundenrechnungen. Für einen Kunden kann es mehrere Rechnungen geben; dann haben alle dieselbe Kundennummer. Das von uns angegebene Sortierverfahren kann diesen Fall (Sortieren nach aufsteigenden Kundennummern) durchaus erledigen, zieht aber aus der Tatsache möglicher Wiederholungen keinen Nutzen.

Man nennt ein Sortierverfahren *glatt* (englisch: *smooth*), wenn es $N$ verschiedene Schlüssel im Mittel in $O(N\log N)$ und $N$ gleiche Schlüssel in $O(N)$ Schritten zu sortieren vermag mit einem „glatten“ Übergang zwischen diesen Werten. Wir ersparen uns eine präzise Definition dieses Begriffs und geben stattdessen ein Beispiel für ein solches Verfahren an.

Die wesentliche Idee besteht darin, bei der Aufteilung eines Bereiches $a[l]\ldots a[r]$ nach dem Schlüssel des rechtesten Elementes $a[r]$ alle Elemente im aufzuteilenden Bereich, deren Schlüssel gleich dem Schlüssel des Pivotelementes sind, in der Mitte zu sammeln. Anstelle einer Zerlegung in zwei Folgen $F_1$ und $F_2$ mit dem Pivotelement dazwischen wird also eine Zerlegung in drei Folgen $F_l$, $F_m$ und $F_r$ angestrebt, sodass gilt

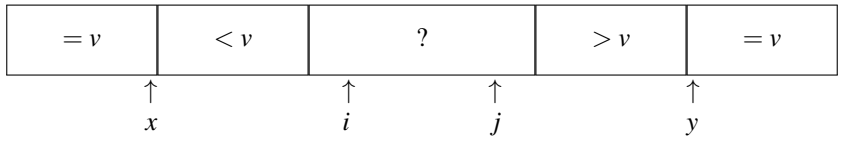
$F_l$ enthält alle Elemente mit Schlüssel $< v$;
$F_m$ enthält alle Elemente mit Schlüssel $= v$;
$F_r$ enthält alle Elemente mit Schlüssel $> v$.

Hier bezeichnet $v = a[r].key$ das Pivotelement.

Da wir natürlich eine In-situ-Aufteilung des Bereichs $a[l]\ldots a[r]$ haben wollen und da wir außerdem nicht im Vorhinein wissen, wo die endgültige Position des Pivotelementes ist, ergibt sich folgendes Problem: Wo soll man die Elemente zwischenspeichern, deren Schlüssel mit dem des Pivotelementes übereinstimmen?

Zur Lösung dieses Problems gibt es (wenigstens) vier verschiedene Möglichkeiten (vgl. [210]). Erstens kann man die Elemente am Anfang und Ende des aufzuteilenden Bereichs sammeln und sie dann später in die Mitte befördern. Zweitens kann man die Elemente nur am Anfang oder nur am Ende sammeln. Drittens kann man sie als wachsenden Block durch das Array wandern lassen, bis sie schließlich ihre richtige Position erreicht haben. Schließlich kann man sie in der ersten oder zweiten Hälfte an vielen Stellen verstreut ablegen und in einem zweiten Durchgang sammeln.

Wir diskutieren nur die erste Möglichkeit genauer. Außer den zwei Zeigern $i$ und $j$, mit denen wir über das Array $a[l]\ldots a[r]$ hinweg wandern, verwenden wir zwei weitere Zeiger $x$ und $y$, die das jeweilige Ende des Anfangs- und Endstücks von $a[l]\ldots a[r]$ markieren, in dem die Elemente mit Schlüssel gleich dem des Pivotelements gesammelt werden.




Anfangs ist $i = l-1$, $j = r$, $x = l-1$, $y = r$. Die Schleife zur Aufteilung des Bereichs $a[l]\ldots a[r]$ nach dem Pivotelement $v = a[r].key$ bekommt jetzt folgende Gestalt:

```
begin-loop
  repeat i := i + 1 until a[i].key ≥ v;
  repeat j := j − 1 until a[j].key ≤ v;
  if i ≥ j then exit-loop;
  if (a[i].key > v) and (a[j].key < v)
    then {vertausche a[i] und a[j]}
      begin
        t := a[i];
        a[i] := a[j];
        a[j] := t
      end;
```

```
    if (a[i].key > v) and (a[j].key = v)
      then {hänge a[j] an das linke Endstück an}
        begin
          t := a[j];
          a[j] := a[i];
          a[i] := a[x+1];
          a[x+1] := t;
          x := x+1
        end;
    if (a[i].key = v) and (a[j].key < v)
      then {hänge a[i] an das rechte Endstück an}
        begin
          t := a[i];
          a[i] := a[j];
          a[j] := a[y-1];
          a[y-1] := t;
          y := y-1
        end;
    if (a[i].key = v) and (a[j].key = v)
      then
        begin
          {hänge a[i] an das linke Endstück an}
          t := a[i];
          a[i] := a[x+1];
          a[x+1] := t;
          x := x+1;
          {hänge a[j] an das rechte Endstück an}
          t := a[j];
          a[j] := a[y-1];
          a[y-1] := t;
          y := y-1
        end
  end-loop
```

Am Ende der Aufteilung steht der Zeiger $i$ auf dem ersten Element des Teilstücks mit Schlüssel größer oder gleich $v$. Dies ist die erste Position, an die das rechte Endstück mit Schlüsseln gleich dem Pivotelement getauscht werden muss; das linke Endstück muss links neben dieser Position zu liegen kommen. Da die Längen aller beteiligten Teilstücke bekannt sind kann man dies leicht mit zwei Schleifen (ohne weitere Schlüsselvergleiche auszuführen) programmieren. Wir überlassen die Einzelheiten dem Leser.

Nach der Aufteilung in die drei Teilfolgen $F_l$, $F_m$ und $F_r$ müssen natürlich nur $F_l$ und $F_r$ rekursiv auf dieselbe Art sortiert werden. Für eine Datei, in der keine Schlüssel mehrfach auftreten, bedeutet das keine Ersparnis. Sind – das ist das andere Extrem – alle Schlüssel identisch, ist überhaupt kein rekursiver Aufruf nötig. L. Wegner [210] zeigt, dass unter geeigneten Annahmen über die Verteilung der Schlüssel gilt, dass das oben skizzierte, auf einem Drei-Wege-Split beruhende Quicksort im Mittel $O(N \log n + N)$ Zeit benötigt, wobei $n$ die Anzahl der verschiedenen Schlüssel unter den $N$ Schlüsseln der Eingabefolge ist.

## 2.3 Heapsort

Alle in den Abschnitten 2.1 und 2.2 behandelten Sortierverfahren benötigen im schlimmsten Fall eine Laufzeit von $\Theta(N^2)$ für das Sortieren von $N$ Schlüsseln. Im Abschnitt 2.8 wird gezeigt, dass zum Sortieren von $N$ Schlüsseln mindestens $\Omega(N \log N)$ Schritte benötigt werden, wenn Information über die Ordnung der Schlüssel nur durch Schlüsselvergleiche gewonnen werden kann. Solche Sortierverfahren heißen *allgemeine* Sortierverfahren, weil außer der Existenz einer Ordnung keine speziellen Bedingungen an die Schlüssel geknüpft sind. Man kann sich nun fragen: Gibt es überhaupt Sortierverfahren, die mit $O(N \log N)$ Operationen auskommen, selbst im schlimmsten Fall? Wir werden sehen, dass solche Verfahren tatsächlich existieren; Heapsort ist eines von ihnen.

*Heapsort* (Sortieren mit einer Halde) folgt dem Prinzip des Sortierens durch Auswahl (vgl. Abschnitt 2.1.1), wobei aber die Auswahl geschickt organisiert ist. Dazu wird eine Datenstruktur verwendet, der *Heap* (die *Halde*), in der die Bestimmung des Maximums einer Menge von $N$ Schlüsseln in einem Schritt möglich ist. Eine Folge $F = k_1, k_2, \ldots, k_N$ von Schlüsseln nennen wir einen Heap, wenn $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ für $2 \leq i \leq N$ gilt. Anders ausgedrückt: $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, sofern $2i \leq N$ bzw. $2i+1 \leq N$.

**Beispiel:** Die Folge $F = 8,6,7,3,4,5,2,1$ genügt der Heap-Bedingung, weil gilt: $8 \geq 6$, $8 \geq 7$, $6 \geq 3$, $6 \geq 4$, $7 \geq 5$, $7 \geq 2$, $3 \geq 1$. Diese Beziehung kann man grafisch wie in Abbildung 2.4 veranschaulichen.

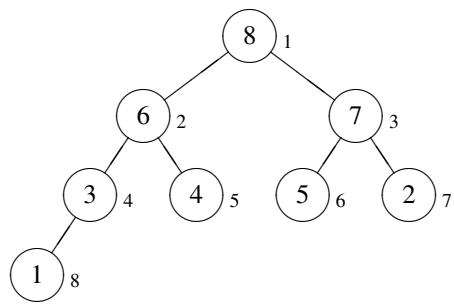



**Abbildung 2.4**

Beim Eintrag $k_i$ ist der Index $i$ mit angegeben um den Bezug zwischen $F$ und dem Schaubild zu erleichtern. In die oberste Zeile kommt der Schlüssel $k_1$; in die nächste Zeile kommen die Schlüssel $k_2$ und $k_3$. Die Beziehungen $k_1 \geq k_2$ und $k_1 \geq k_3$ werden durch zwei Verbindungslinien (Kanten) dargestellt. In Zeile $j$ kommen Schlüssel $k_{2^{j-1}}$ bis $k_{2^j-1}$, von links nach rechts. Außerdem werden Kanten zu den entsprechenden Schlüsseln der vorangehenden Zeile gezeichnet. Das so definierte Schaubild repräsentiert den Heap als Binärbaum (vgl. hierzu auch Kapitel 5). Jedem Schlüssel entspricht

ein Knoten des Baumes und zwischen den Knoten für Schlüssel $k_i$ und $k_{2i}$ bzw. $k_i$ und $k_{2i+1}$ gibt es eine Kante. Schlüssel $k_1$ steht an der Wurzel des Baumes. Schlüssel $k_{2i}$ ist der linke, $k_{2i+1}$ der rechte Sohn von Schlüssel $k_i$; $k_i$ ist der Vater von $k_{2i}$ und $k_{2i+1}$. Interpretiert man den Heap als Binärbaum, so kann man die Heap-Bedingung auch wie folgt formulieren. Ein Binärbaum ist ein Heap, wenn der Schlüssel jedes Knotens mindestens so groß ist wie die Schlüssel seiner beiden Söhne (falls es diese gibt).

Wir gehen im Folgenden immer davon aus, dass die Schlüssel in einem Array gespeichert sind, auch wenn wir manchmal in Erklärungen auf die Baumstruktur Bezug nehmen. Stellen wir uns einmal vor, eine Folge von Schlüsseln sei als Heap gegeben (also etwa Folge $F$ im obigen Beispiel) und wir sollen die Schlüssel in absteigender Reihenfolge ausgeben. Das ist für den ersten Schlüssel ganz leicht, denn $k_1$ ist ja das Maximum aller Schlüssel. Wie bestimmen wir aber jetzt den nächstkleineren Schlüssel? Eine offensichtliche Methode ist doch die, den gerade ausgegebenen Schlüssel aus der Folge zu entfernen und die restliche Folge wieder zu einem Heap zu machen. Dann steht nämlich der nächstkleinere Schlüssel wieder an der Wurzel und wir können nach demselben Verfahren fortfahren.

Das ergibt für das absteigende Sortieren der Schlüssel eines Heaps folgende **Methode:**

*{Anfangs besteht der Heap aus Schlüsseln $k_1, \ldots, k_N$}*
*Solange der Heap nicht leer ist, wiederhole:*
  *gib $k_1$ aus; {das ist der nächstgrößere Schlüssel}*
  *entferne $k_1$ aus dem Heap;*
  *stelle die Heap-Bedingung für die restlichen Schlüssel her,*
    *sodass die neue Wurzel an Position 1 steht.*

Der schwierigste Teil ist hier das Wiederherstellen der Heap-Bedingung. Wir nutzen die Tatsache aus, dass nach dem Entfernen der Wurzel ja noch zwei Teil-Heaps vorliegen. Im obigen Beispiel der Folge $F = 8,6,7,3,4,5,2,1$ gibt es nach Entfernen von $k_1 = 8$ zwei Teil-Heaps, die Abbildung 2.5 zeigt.

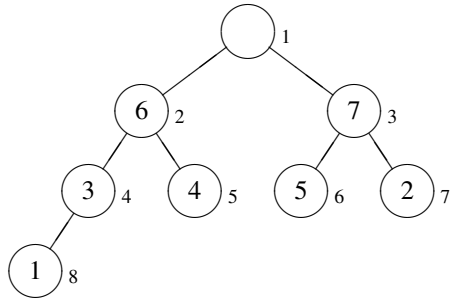



**Abbildung 2.5**

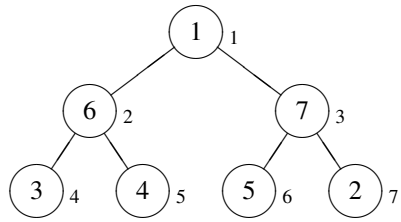



**Abbildung 2.6**

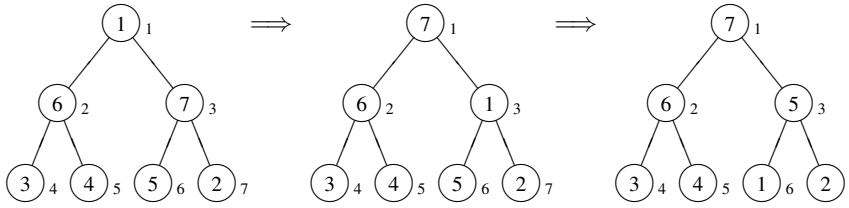



**Abbildung 2.7**

Wir machen daraus einen Heap, indem wir zunächst den Schlüssel mit höchstem Index an die Wurzel schreiben, wobei aber im Allgemeinen die Heap-Bedingung verletzt wird. Dies zeigt Abbildung 2.6.

Dann lassen wir den (neuen) Schlüssel $k_1$ im Heap nach unten *versickern* (*sift down*), indem wir ihn solange immer wieder mit dem größeren seiner beiden Söhne vertauschen, bis beide Söhne kleiner sind oder der Schlüssel unten angekommen ist, vgl. Abbildung 2.7.

Damit ist die Heap-Bedingung für die Schlüsselfolge erfüllt. Für das Entfernen des Maximums und das Herstellen der Heap-Bedingung für die Folge der Schlüssel $k_1, \ldots, k_m$ eignet sich also die folgende Methode:

*{entferne Maximum aus Heap $k_1, \ldots, k_m$ und mache*
*restliche Schlüsselfolge wieder zu einem Heap}*
*übertrage $k_m$ nach $k_1$ ;*
*versickere $k_1$ im Bereich $k_1$ bis $k_{m-1}$.*

Das Versickern eines Schlüssels geschieht wie folgt:

*{versickere $k_i$ im Bereich $k_i$ bis $k_m$}*
*Solange $k_i$ einen linken Sohn $k_j$ hat, wiederhole:*
*falls $k_i$ einen rechten Sohn hat,*
*so sei $k_j$ derjenige Sohn von $k_i$ mit größerem Schlüssel;*
*falls $k_i < k_j$,*
*so vertausche $k_i$ mit $k_j$ und setze $i := j$,*
*sonst halte an {die Heap-Bedingung gilt}.*

Tabelle 2.2 zeigt am Beispiel der Folge $F = 8,6,7,3,4,5,2,1$, wie die Schlüssel von $F$ absteigend sortiert werden.

| Kommentar | Ausgabe | $k_1$ | $k_2$ | $k_3$ | $k_4$ | $k_5$ | $k_6$ | $k_7$ | $k_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Anfangsheap | | 8 | 6 | 7 | 3 | 4 | 5 | 2 | 1 |
| gib $k_1$ aus | 8 | | | | | | | | |
| übertrage $k_8$ nach $k_1$ | | 1 | 6 | 7 | 3 | 4 | 5 | 2 | |
| versickere $k_1$ | | 7 | | 1 | | | | | |
| | | | | 5 | | | 1 | | |
| gib $k_1$ aus | 7 | | | | | | | | |
| übertrage $k_7$ nach $k_1$ | | 2 | 6 | 5 | 3 | 4 | 1 | | |
| versickere $k_1$ | | 6 | 2 | | | | | | |
| | | | 4 | | | 2 | | | |
| gib $k_1$ aus, übertrage $k_6$, | 6 | 5 | 4 | 1 | 3 | 2 | | | |
| versickere etc. | 5 | 4 | 3 | 1 | 2 | | | | |
| | 4 | 3 | 2 | 1 | | | | | |
| | 3 | 2 | 1 | | | | | | |
| | 2 | 1 | | | | | | | |
| | 1 | leer | | | | | | | |

**Tabelle 2.2**

Statt die Schlüssel in absteigender Reihenfolge auszugeben, können wir sie mit dem angegebenen Verfahren auch in aufsteigender Reihenfolge sortieren, wenn wir das jeweils aus dem Heap entfernte Maximum nicht ausgeben, sondern an die Stelle desjenigen Schlüssels schreiben, der nach dem Entfernen des Maximums nach $k_1$ übertragen wird. Dann lässt sich das Sortieren eines Heaps wie folgt beschreiben:

```
{sortiere Heap a : sequence im Bereich von 1 bis r : integer}
var
  i : integer;
  t : item;
begin
  for i := r downto 2 do
    begin {tausche a[1] mit a[i], versickere a[1]}
{M1} t := a[i];
{M1} a[i] := a[1];
{M1} a[1] := t;
     versickere(a,1,i-1)
    end
end
```

Dabei ist *versickere* wie folgt erklärt:

```
procedure versickere (var a : sequence; i, m: integer);
{versickere a[i] bis höchstens nach a[m]}
var
  j : integer;
  t : item;
begin
  while 2 * i ≤ m do {a[i] hat linken Sohn}
    begin
      j := 2 * i; {a[j] ist linker Sohn von a[i]}
      if j < m
        then {a[i] hat rechten Sohn}
{C1}      if a[j].key < a[j+1].key then j := j+1;
      { jetzt ist a[j].key größer}
{C2}  if a[i].key < a[j].key
        then {tausche a[i] mit a[j]}
          begin
{M2}        t := a[i];
{M2}        a[i] := a[j];
{M2}        a[j] := t;
            i := j {versickere weiter}
          end
        else i := m {halte an, Heap-Bedingung erfüllt}
    end
end
```

**Analyse:** Außerhalb der Prozedur *versickere* werden beim Sortieren eines Heaps, der aus $N$ Schlüsseln besteht, gerade $\Theta(N)$ Bewegungen von Datensätzen ausgeführt (vgl. Programmzeilen $\{M1\}$). Außerdem werden beim Versickern Datensätze bewegt (vgl. Programmzeilen $\{M2\}$). Beim Versickern wird ein Schlüssel wiederholt mit einem seiner Söhne vertauscht. Im Schaubild, das den Heap als Binärbaum zeigt, wandert der Schlüssel bei jeder Vertauschung eine Zeile – man sagt: eine *Stufe* oder ein *Niveau* (*level*) – tiefer. Die Anzahl der Schlüssel verdoppelt sich von Stufe zu Stufe; lediglich auf der letzten Stufe können einige Schlüssel fehlen. Ein Heap mit $j$ Stufen speichert also zwischen $2^{j-1}$ und $2^j - 1$ Schlüssel. Ein Heap für $N$ Schlüssel, mit $2^{j-1} \leq N \leq 2^j - 1$, hat also $j = \lceil \log(N+1) \rceil$ Stufen. Daher kann die **while**-Schleife der Prozedur *versickere* bei einem Prozeduraufruf höchstens $\lceil \log(N+1) \rceil - 1$ Mal durchlaufen werden. Da die Prozedur *versickere* genau $N-1$ Mal aufgerufen wird, ergibt sich eine obere Schranke von $O(N \log N)$ Ausführungen jeder der Zeilen $\{C1\}$, $\{C2\}$ und $\{M2\}$. Damit gilt:

$$C_{max}(N) = O(N \log N), \;\; M_{max}(N) = O(N \log N).$$

Das Verfahren einen Heap zu sortieren können wir erst dann zum Sortieren einer beliebigen Schlüsselfolge verwenden, wenn wir diese in einen Heap umgewandelt haben. Der Erfinder von Heapsort, J. W. J. Williams (vgl. [212]), hat dafür eine Methode angegeben, die in $O(N \log N)$ Schritten einen Heap konstruiert. Ein schnelleres Verfahren, das wir im Folgenden erläutern, stammt von R. W. Floyd (vgl. [60]).

Die Grundidee besteht darin, in einer Schlüsselfolge von hinten nach vorne Teil-Heaps zu erzeugen. Nehmen wir an, die Heap-Bedingung sei für alle Schlüssel der

Folge ab einem gewissen $k_l$ erfüllt, d. h., es gelte $k_{\lfloor \frac{i}{2} \rfloor} \geq k_i$ für $\lfloor \frac{i}{2} \rfloor \geq l$. Das ist anfangs, in der unsortierten Folge, gesichert für $l = \lfloor \frac{N}{2} \rfloor + 1$. Dann können wir die Heap-Bedingung für alle Schlüssel ab $k_{l-1}$ herstellen, indem wir $k_{l-1}$ in der Folge $k_{l-1}, \ldots, k_N$ versickern. Die Voraussetzung für das Versickern eines Schlüssels, nämlich, dass die beiden Söhne des Schlüssels Wurzeln von Teil-Heaps sind, ist gesichert, weil die Heap-Bedingung für alle Schlüssel mit höherem Index erfüllt ist. Zunächst lassen wir also $k_{\lfloor \frac{N}{2} \rfloor}$ versickern, dann $k_{\lfloor \frac{N}{2} \rfloor - 1}$ usw., bis schließlich $k_1$ versickert. Die erhaltene Folge ist ein Heap, weil die Heap-Bedingung ab Schlüssel $k_1$, also für alle Schlüssel, erfüllt ist.

**Methode:** Eine gegebene Folge $F = k_1, k_2, \ldots, k_N$ von $N$ Schlüsseln wird in einen Heap umgewandelt, indem die Schlüssel $k_{\lfloor \frac{N}{2} \rfloor}$, $k_{\lfloor \frac{N}{2} \rfloor - 1}$, …, $k_1$ (in dieser Reihenfolge) in $F$ versickern.

**Beispiel:** Betrachten wir die Folge $F = 2, 1, 5, 3, 4, 8, 7, 6$ und die Veränderungen, die sich beim Versickern der Schlüssel ergeben:

| Versickere Schlüssel | Folge |
|---|---|
| anfangs | 2, 1, 5, 3, 4, 8, 7, 6 |
| $k_4 = 3$ | 2, 1, 5, 6, 4, 8, 7, 3 |
| $k_3 = 5$ | 2, 1, 8, 6, 4, 5, 7, 3 |
| $k_2 = 1$ | 2, 6, 8, 3, 4, 5, 7, 1 |
| $k_1 = 2$ | 8, 6, 7, 3, 4, 5, 2, 1 |

Die erhaltene Folge ist ein Heap.

Das Sortierverfahren Heapsort für eine Folge $F$ von Schlüsseln besteht nun darin, $F$ zunächst in einen Heap umzuwandeln und den Heap dann zu sortieren. Das ergibt folgende Sortierprozedur.

```
procedure heapsort (var a : sequence);
{sortiert die Elemente a[1] bis a[N]}
var
  i : integer;
  t : item;
begin
  {wandle a[1] bis a[N] in einen Heap um}
  for i := N div 2 downto 1 do versickere(a,i,N);
  {sortiere den Heap}
  for i := N downto 2 do
    begin {tausche a[1] mit a[i], versickere a[1]}
      t := a[i];
      a[i] := a[1];
      a[1] := t;
      versickere(a,1,i-1)
    end
end
```

**Analyse:** Sei $2^{j-1} < N \leq 2^j - 1$, also $j$ die Anzahl der Stufen des Heaps für $N$ Schlüssel. Nummerieren wir die Stufen von oben nach unten von 1 bis $j$. Dann gibt es auf Stufe $k$ höchstens $2^{k-1}$ Schlüssel. Die Anzahl der Bewege- und Vergleichsoperationen zum Versickern eines Elementes der Stufe $k$ ist proportional zu $j-k$. Insgesamt ergibt sich für die Anzahl der Operationen zum Umwandeln einer unsortierten Folge in einen Heap:

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) = \sum_{k=1}^{j-1} k \cdot 2^{j-k-1} = 2^{j-1} \sum_{k=1}^{j-1} \frac{k}{2^k} \leq N \cdot 2 = O(N)$$

Das Aufbauen eines Heaps aus einer unsortierten Folge ist also in linearer Zeit möglich. Damit ergibt sich die Zeitschranke für Heapsort aus dem Sortieren des Heaps zu

$$C_{max}(N) = O(N \log N), \;\; M_{max}(N) = O(N \log N).$$

Experimente zeigen, dass dies auch die mittlere Anzahl von Bewegungen und Vergleichsoperationen für Heapsort ist. Heapsort ist also das erste von uns behandelte Sortierverfahren, das asymptotisch optimale Laufzeit im schlechtesten Fall hat. Diese Laufzeit variiert für verschiedene Eingabefolgen nur geringfügig; insbesondere nützt oder schadet Vorsortierung bei Heapsort praktisch nichts. Man kann Heapsort jedoch so modifizieren, dass es Vorsortierung ausnützt. Eine solche Heapsort-Variante, *Smoothsort* [41], benötigt $O(N)$ Zeit für eine vorsortierte Folge und $O(N \log N)$ Zeit im schlimmsten Fall. Heapsort ist kein *stabiles* Verfahren, d. h., die relative Position gleicher Schlüssel kann sich beim Sortieren ändern. Im Gegensatz zu den gängigen Varianten von Quicksort, das im Durchschnitt schneller ist als Heapsort, benötigt Heapsort nur konstant viel zusätzlichen Speicherplatz; es ist also ein echtes In-situ-Sortierverfahren.

## 2.4 Mergesort

Das Verfahren *Mergesort* (*Sortieren durch Verschmelzen*) ist eines der ältesten und bestuntersuchten Verfahren zum Sortieren mithilfe von Computern. John von Neumann hat es bereits 1945 vorgeschlagen. Es folgt – ähnlich wie Quicksort – der Strategie eine Folge durch rekursives Aufteilen zu sortieren. Im Unterschied zu Quicksort wird aber hier die Folge in gleich große Teilfolgen aufgeteilt. Die (rekursiv) sortierten Teilfolgen werden dann verschmolzen. Dazu verwendet man linear viel zusätzlichen Speicherplatz. Als Ausgleich dafür kann die Laufzeit von Mergesort für eine Folge von $N$ Sätzen $O(N \log N)$ nicht übersteigen.

In Abschnitt 2.4.1 beschreiben und analysieren wir eine einfache Realisierung von Mergesort, das rekursive Aufteilen in zwei Teilfolgen (*2-Wege-Mergesort*). Man kann Mergesort auch leicht ohne Rekursion als das Verschmelzen immer größerer Teilfolgen formulieren; dieses *reine 2-Wege-Mergesort* (*straight 2-way merge sort*) beschreiben wir im Abschnitt 2.4.2. Nützt man die in der zu sortierenden Folge bereits vorhandenen sortierten Teilfolgen aus, so erhält man das *natürliche 2-Wege-Mergesort* (*natural 2-way merge sort*); dieses Verfahren beschreiben wir im Abschnitt 2.4.3. Schließlich eignet sich Mergesort ganz besonders gut für das Sortieren von Daten auf Sekundärspeichern, das *externe Sortieren* (vgl. Abschnitt 2.7).

### 2.4.1 2-Wege-Mergesort

**Methode:** Eine Folge $F = k_1, \ldots, k_N$ von $N$ Schlüsseln wird sortiert, indem sie zunächst in zwei möglichst gleich große Teilfolgen $F_1 = k_1, \ldots, k_{\lceil \frac{N}{2} \rceil}$ und $F_2 = k_{\lceil \frac{N}{2} \rceil + 1}, \ldots, k_N$ aufgeteilt wird. Dann wird jede dieser Teilfolgen mittels Mergesort sortiert. Die sortierte Folge ergibt sich durch Verschmelzen der beiden sortierten Teilfolgen. Mergesort folgt also, ähnlich wie Quicksort, dem allgemeinen Prinzip des Divide-and-conquer. Dabei ist wichtig, dass das Verschmelzen sortierter Folgen einfacher ist als das Sortieren. Zwei sortierte Folgen werden verschmolzen, indem man je einen Positionszeiger (Index) durch die beiden Folgen so wandern lässt, dass die Elemente beider Folgen insgesamt in sortierter Reihenfolge angetroffen werden. Beginnt man mit beiden Zeigern am jeweils linken Ende der beiden Folgen, so bewegt man in einem Schritt denjenigen der beiden Zeiger um eine Position nach rechts (in der betreffenden Folge), der auf den kleineren Schlüssel zeigt. Man übernimmt einen Schlüssel immer dann in die Resultatfolge, wenn ein Zeiger bisher auf diesen Schlüssel gezeigt hat und im aktuellen Schritt weiterwandert. Sobald eine der Folgen erschöpft ist, übernimmt man den Rest der anderen Folge in die Resultatfolge.

**Beispiel:** Die beiden sortierten Folgen $F_1 = 1,2,3,5,9$ und $F_2 = 4,6,7,8,10$ sollen verschmolzen werden. Zunächst zeigen zwei Positionszeiger $i$ und $j$ auf die Anfangselemente beider Folgen, also 1 und 4, wie in Abbildung 2.8 in der ersten Zeile dargestellt.

| | $F_1$ | $F_2$ | Resultatfolge |
|---|---|---|---|
| | 1, 2, 3, 5, 9 | 4, 6, 7, 8, 10 | |
| anfangs: | $\uparrow i$ | $\uparrow j$ | — |
| $1 < 4$: | $\uparrow i$ | $\uparrow j$ | 1 |
| $2 < 4$: | $\uparrow i$ | $\uparrow j$ | 1, 2 |
| $3 < 4$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3 |
| $5 > 4$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4 |
| $5 < 6$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4, 5 |
| $9 > 6$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4, 5, 6 |
| $9 > 7$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4, 5, 6, 7 |
| $9 > 8$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4, 5, 6, 7, 8 |
| $9 < 10$: | $\uparrow i$ | $\uparrow j$ | 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| $F_1$ erschöpft; | | $\uparrow j$ | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |

$F_2$ erschöpft: Stopp.

**Abbildung 2.8**

Da $k_i < k_j$ gilt, wandert Zeiger $i$ in Folge $F_1$ und $k_i$ wird in die Resultatfolge übernommen (mit „$1 < 4$ :“ beschriftete Zeile).

Im nächsten Schritt ist wieder $k_i = 2 < 4 = k_j$, also wandert wieder Zeiger $i$ in Folge $F_1$. Wir zeigen im Rest der Abbildung 2.8 den Prozess des Verschmelzens bis zum Ende.

Die Struktur des Verfahrens Mergesort kann man, ohne Berücksichtigung von Implementationsdetails, wie folgt beschreiben:

**Algorithmus** *Mergesort*$(F : Folge)$;
{*sortiert Schlüsselfolge F nach aufsteigenden Werten*}
*Falls F die leere Folge ist oder nur aus einem einzigen Schlüssel besteht, bleibt F unverändert; sonst:*

*Divide*: *Teile F in zwei etwa gleich große Teilfolgen,* $F_1$ *und* $F_2$*;*
*Conquer*: *Mergesort*$(F_1)$*; Mergesort*$(F_2)$*;*
{*jetzt sind beide Teilfolgen* $F_1$ *und* $F_2$ *sortiert*}
*Merge*: *Bilde die Resultatfolge durch Verschmelzen von* $F_1$ *und* $F_2$.

Betrachten wir als Beispiel die Anwendung des Verfahrens Mergesort auf die Folge $F = 2, 1, 3, 9, 5, 6, 7, 4, 8, 10$. Zunächst wird $F$ aufgeteilt in die beiden Teilfolgen $F_1 = 2, 1, 3, 9, 5$ und $F_2 = 6, 7, 4, 8, 10$. Dann werden beide Teilfolgen mittels Mergesort sortiert; das ergibt $F_1 = 1, 2, 3, 5, 9$ und $F_2 = 4, 6, 7, 8, 10$. Diese beiden Folgen werden, wie im vorangegangenen Beispiel gezeigt, zur Folge $F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ verschmolzen.

Für die programmtechnische Realisierung von Mergesort nehmen wir an, dass die Folge der zu sortierenden Datensätze in einem Feld $a$ an den Positionen 1 bis $N$ gespeichert ist. Weil *mergesort* rekursiv für Teilfolgen aufgerufen wird, verwenden wir zwei Feldindizes für das erste und das letzte Element der zu sortierenden Teilfolge.

```
procedure mergesort (var a : sequence; l, r : integer);
{sortiert a[l] bis a[r] nach aufsteigenden Schlüsseln}
var
  m : integer;
begin
  if l < r {sonst : leere oder einelementige Folge}
    then
      begin
        m := (l + r) div 2;  {das ist die Mitte der Folge}
        mergesort(a, l, m);
        mergesort(a, m + 1, r);
        {a[l] ... a[m] und a[m + 1] ... a[r] sind sortiert}
        merge(a, l, m, r)  {Verschmelzen}
      end
end
```

Das Verschmelzen zweier Teilfolgen, die im Feld $a$ an benachbarten Feldpositionen stehen, wird durch die Prozedur *merge* erreicht. Wir verwenden dazu ein zusätzliches Feld $b$, das zunächst die Resultatfolge aufnimmt. Anschließend wird die Resultatfolge von $b$ nach $a$ zurückkopiert.

```
        procedure merge (var a : sequence; l, m, r : integer);
        {verschmilzt die beiden sortierten Teilfolgen a[l]...a[m]
         und a[m+1]...a[r] und speichert sie in a[l]...a[r]}
        var
          b : sequence; {Hilfsfeld zum Verschmelzen}
          h, i, j, k : integer;
        begin
          i := l; {inspiziere noch a[i] bis a[m] der ersten Teilfolge}
          j := m+1; {inspiziere noch a[j] bis a[r] der zweiten Teilfolge}
          k := l; {das nächste Element der Resultatfolge ist b[k]}
          while (i ≤ m) and (j ≤ r) do
            begin {beide Teilfolgen sind noch nicht erschöpft}
{C}           if a[i].key ≤ a[j].key
                then {übernimm a[i] nach b[k]}
                  begin
{M1}                b[k] := a[i];
                    i := i+1
                  end
                else {übernimm a[j] nach b[k]}
                  begin
{M1}                b[k] := a[j];
                    j := j+1
                  end;
              k := k+1
            end;
          if i > m
          then {erste Teilfolge ist erschöpft; übernimm zweite}
{M2}        for h := j to r do b[k+h-j] := a[h]
          else {zweite Teilfolge ist erschöpft; übernimm erste}
{M2}        for h := i to m do b[k+h-i] := a[h];
          {speichere sortierte Folge von b zurück nach a}
{M3}      for h := l to r do a[h] := b[h]
        end
```

Man erkennt, dass die für beide Teilfolgen erforderlichen Aktionen völlig gleichartig sind; wie man diese Aktionen parametrisiert, beschreiben wir beim Verschmelzen mehrerer Teilfolgen in Abschnitt 2.7.

**Analyse:** Schlüsselvergleiche werden nur in der Prozedur *merge* in der mit $\{C\}$ markierten Zeile ausgeführt. Nach jedem Schlüsselvergleich wird einer der beiden Positionszeiger weiterbewegt. Sobald eine Teilfolge erschöpft ist, werden keine weiteren Schlüsselvergleiche mehr ausgeführt. Für zwei Teilfolgen der Länge $n_1$ bzw. $n_2$ ergeben sich also mindestens $min(n_1,n_2)$ und höchstens $n_1+n_2-1$ Schlüsselvergleiche. Zum Verschmelzen zweier etwa gleich langer Teilfolgen der Gesamtlänge $N$ benötigen wir also $\Theta(N)$ Schlüsselvergleiche; das ist der ungünstigste Fall. Damit ergibt sich für

die Anzahl $C(N)$ der zum Sortieren von $N$ Schlüsseln benötigten Vergleichsoperationen

$$C(N) = \underbrace{C\left(\left\lceil \frac{N}{2} \right\rceil\right) + C\left(\left\lfloor \frac{N}{2} \right\rfloor\right)}_{\text{Schlüsselvergleiche zum Sortieren der beiden Teilfolgen}} + \underbrace{\Theta(N)}_{\text{Verschmelzen}} = \Theta(N \log N).$$

Das gilt für den besten ebenso wie für den schlechtesten (und damit auch für den mittleren) Fall gleichermaßen:

$$C_{min}(N) = C_{max}(N) = C_{mit}(N) = \Theta(N \log N).$$

Mergesort ist also ein Sortierverfahren, das größenordnungsmäßig nicht mehr Schlüsselvergleiche benötigt, als im schlimmsten Fall auch tatsächlich erforderlich sind (vgl. Abschnitt 2.8); es ist worst-case-optimal. Damit hat es sich ausgezahlt die rekursive Aufteilung möglichst ausgeglichen vorzunehmen. Da bei jedem Aufteilungsschritt die Folgenlänge etwa halbiert wird, ergeben sich nach $\lceil \log N \rceil$ Aufteilungsschritten stets Teilfolgen der Länge 1, die nicht weiter rekursiv behandelt werden müssen. Die Rekursionstiefe ist also – etwa im Gegensatz zu Quicksort – logarithmisch beschränkt.

An den mit $\{M \ldots\}$ markierten Zeilen der Prozedur *merge* lässt sich ablesen, dass viel mehr Bewegungen von Datensätzen ausgeführt werden als Schlüsselvergleiche. Für jeden Schlüsselvergleich wird auch eine Bewegung eines Datensatzes (Zeilen $\{M1\}$) ausgeführt. Zusätzlich werden die restlichen Elemente einer Teilfolge nach $b$ übernommen (Zeilen $\{M2\}$), wenn die andere Teilfolge erschöpft ist. Schließlich wird noch die gesamte Resultatfolge von $b$ nach $a$ zurückkopiert (Zeile $\{M3\}$). Beim Verschmelzen zweier Teilfolgen der Gesamtlänge $N$ werden also gerade $2N$ Bewegungen ausgeführt. Damit ergibt sich auch hier

$$M_{min}(N) = M_{max}(N) = M_{mit}(N) = \Theta(N \log N).$$

Viele der Bewegungen kann man vermeiden, wenn man den Bereich, in dem sortierte Teilfolgen gespeichert sind (also $a$ oder $b$), parametrisiert (vgl. Abschnitt 2.7). Am asymptotischen Aufwand ändert sich dabei nichts. Weil bei Mergesort Teilfolgen immer nur sequenziell inspiziert werden, kann man dieses Verfahren auch für Datensätze in verketteten linearen Listen verwenden. Dann entfallen Bewegungen von Datensätzen komplett; stattdessen werden lediglich Listenzeiger geändert. Allerdings benötigt auch diese Mergesort-Variante linear viel zusätzlichen Speicherplatz, nämlich für die Listenzeiger.

## 2.4.2 Reines 2-Wege-Mergesort

Im rekursiven 2-Wege-Mergesort, wie in Abschnitt 2.4.1 beschrieben, dient die Prozedur *mergesort* lediglich zur Organisation der Verschmelzungen von Teilfolgen. Das eigentliche Sortieren ist dann erst das Verschmelzen von Teilfolgen der Länge 1, später der Länge 2 usw., bis zum Verschmelzen zweier Teilfolgen der Länge $N/2$. Beim *reinen 2-Wege-Mergesort* (*straight 2-way merge sort*) werden die Verschmelzungen von Teilfolgen genauso organisiert, und zwar ohne Rekursion und ohne Aufteilung.

**Methode:** Eine Folge $F = k_1, \ldots, k_N$ von Schlüsseln wird sortiert, indem sortierte Teilfolgen zu immer längeren Teilfolgen verschmolzen werden. Anfangs ist jeder Schlüssel $k_i$, $1 \leq i \leq N$, eine sortierte Teilfolge. In einem Durchgang (von links nach rechts durch die Folge) werden jeweils zwei benachbarte Teilfolgen zu einer Folge verschmolzen. Beim ersten Durchgang wird also $k_1$ mit $k_2$ verschmolzen, $k_3$ mit $k_4$ usw. Dabei kann es vorkommen, dass am Ende eines Durchganges eine Teilfolge übrig bleibt, die nicht weiter verschmolzen wird. Bei jedem Durchgang verdoppelt sich also die Länge der sortierten Teilfolgen, außer eventuell am rechten Rand. Die gesamte Folge ist sortiert, sobald in einem Durchgang nur noch zwei Teilfolgen verschmolzen worden sind.

**Beispiel:** Betrachten wir die Folge $F = 2,1,3,9,5,6,7,4,8,10$ aus Abschnitt 2.4.1. Teilfolgen sind voneinander durch einen senkrechten Strich getrennt. Anfangs haben alle Teilfolgen die Länge 1.

$$2 \mid 1 \mid 3 \mid 9 \mid 5 \mid 6 \mid 7 \mid 4 \mid 8 \mid 10$$

Nach einem Durchgang sind je zwei benachbarte Teilfolgen verschmolzen.

$$1\,,2 \mid 3\,,9 \mid 5\,,6 \mid 4\,,7 \mid 8\,,10$$

Wir geben die Teilfolgen nach jedem weiteren Durchgang an.

$$\begin{array}{cccc|cccc|cc} 1, & 2, & 3, & 9 \mid & 4, & 5, & 6, & 7 \mid & 8, & 10 \\ 1, & 2, & 3, & 4, & 5, & 6, & 7, & 9 \mid & 8, & 10 \\ 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10 \end{array}$$

Die folgende Prozedur *straightmergesort* realisiert das reine 2-Wege-Mergesort.

```
procedure straightmergesort (var a : sequence; l, r : integer);
  {sortiert a[l]...a[r] nach aufsteigenden Schüsselwerten;
   l und r werden nicht für rekursive Aufrufe benötigt und sind
   nur wegen der Analogie zu mergesort hier angegeben}
  var
    size, ll, mm, rr : integer;
  begin
    size := 1; {Länge der bereits sortierten Teilfolgen}
    while size < r - l + 1 do
      begin {verschmilz Teilfolgen der Länge size}
        rr := l - 1; {Elemente bis inklusive a[rr] sind bearbeitet}
        while rr + size < r do
          begin {es gibt noch mindestens zwei Teilfolgen}
            ll := rr + 1; {linker Rand der ersten Teilfolge}
            mm := ll + size - 1; {rechter Rand}
            if mm + size ≤ r
              then {r noch nicht überschritten}
                rr := mm + size
              else {zweite Teilfolge ist kürzer}
```

```
                rr := r;
              merge(a, ll, mm, rr)
            end;
          {ein Durchlauf ist beendet; sortierte Teilfolgen haben jetzt
           die Länge 2 ∗ size}
          size := 2 ∗ size
        end
      {a ist sortiert}
    end
```

**Analyse:** Schlüsselvergleiche und Bewegungen finden auch hier nur innerhalb der Prozedur *merge* statt. Bei jedem Durchgang durch die Folge werden insgesamt $N$ Datensätze mittels *merge* verschmolzen. Weil sich bei jedem Durchgang die Länge der sortierten Teilfolgen verdoppelt, erhält man nach $\lceil \log N \rceil$ Durchgängen eine sortierte Folge. Damit gilt wie erwartet

$$C_{min}(N) = C_{max}(N) = C_{mit}(N) = \Theta(N \log N)$$

und

$$M_{min}(N) = M_{max}(N) = M_{mit}(N) = \Theta(N \log N).$$

### 2.4.3 Natürliches 2-Wege-Mergesort

Ausgehend vom reinen 2-Wege-Mergesort liegt es nahe den Verschmelze-Prozess nicht mit einelementigen Teilfolgen zu beginnen, sondern bereits anfangs möglichst lange sortierte Teilfolgen zu verwenden. Auf diese Weise versucht man, eine natürliche in der gegebenen Folge bereits enthaltene Vorsortierung auszunutzen. Betrachten wir noch einmal die Folge $F = 2,1,3,9,5,6,7,4,8,10$. In $F$ findet man vier längstmögliche, bereits sortierte Teilfolgen benachbarter Folgenelemente.

$$2 \mid 1,3,9 \mid 5,6,7 \mid 4,8,10$$

Verschmilzt man nun, wie beim reinen 2-Wege-Mergesort, in jedem Durchgang benachbarte Teilfolgen, so erhält man nach zwei Durchgängen eine sortierte Folge.

$$\begin{array}{cccccccccc} 1, & 2, & 3, & 9 \mid & 4, & 5, & 6, & 7, & 8, & 10 \\ 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10 \end{array}$$

**Methode:** Eine Folge $F = k_1, \ldots, k_N$ von Schlüsseln wird sortiert, indem sortierte Teilfolgen zu immer längeren sortierten Teilfolgen verschmolzen werden. Anfangs wird $F$ in längstmögliche sortierte Teilfolgen benachbarter Schlüssel (die so genannten *Runs*) geteilt. Dann werden wiederholt benachbarte Teilfolgen verschmolzen (wie beim reinen 2-Wege-Mergesort), bis schließlich eine sortierte Folge entstanden ist.

Bei der programmtechnischen Realisierung ist der einzige Unterschied zum reinen 2-Wege-Mergesort das Finden der sortierten Teilfolgen. Eine sortierte Teilfolge ist zu Ende, wenn ein kleinerer Schlüssel auf einen größeren folgt. Damit ergibt sich die Prozedur *naturalmergesort*:

```
    procedure naturalmergesort (var a : sequence; l, r : integer);
    {sortiert a[l]...a[r] nach aufsteigenden Schüsselwerten}
    var
      ll, mm, rr : integer;
    begin
      repeat
        rr := l − 1; {Elemente bis inklusive a[rr] sind bearbeitet}
        while rr < r do
          begin {finde und verschmilz die nächsten Runs}
            ll := rr + 1; {linker Rand}
            mm := ll; {a[ll]...a[mm] ist sortiert}
{C1}        while (mm < r) and (a[mm + 1].key ≥ a[mm].key) do
              mm := mm + 1;
            {jetzt ist mm das letzte Element des ersten Runs}
            if mm < r
              then {es ist noch ein zweiter Run vorhanden}
                begin
                  rr := mm + 1; {rechter Rand}
{C1}              while (rr < r) and (a[rr + 1].key ≥ a[rr].key) do
                    rr := rr + 1;
                  merge(a, ll, mm, rr)
                end
              else {kein zweiter Run vorhanden: fertig}
                rr := mm
          end
      until ll = l {dann ist a[l]...a[r] ein Run, also sortiert}
    end
```

Die angegebene Prozedur *naturalmergesort* ist so noch nicht ganz korrekt. Die beiden kombinierten Bedingungen

$$(mm < r) \textbf{ and } (a[mm+1].key \geq a[mm].key)$$

und

$$(rr < r) \textbf{ and } (a[rr+1].key \geq a[rr].key)$$

führen zu einem Fehler, wenn das Feldelement $a[r+1]$ nicht existiert. Der Grund liegt darin, dass in Pascal keine Annahmen über das Auswerten von Teilen zusammengesetzter Bedingungen gemacht werden. Das bedeutet, dass möglicherweise der Teil $(a[mm+1].key \geq a[mm].key)$ der ersten Bedingung auch dann noch ausgewertet wird, wenn $(mm < r)$ bereits den Wert *false* liefert. Dann ist aber $mm = r$ und damit erfolgt ein Zugriff auf $a[r+1]$. Der Wert des Feldelements beeinflusst den Wahrheitswert der die Schleife kontrollierenden Bedingung natürlich nicht. Es genügt also ein um ein Feldelement größeres Feld zu vereinbaren und das Feldelement mit dem höchsten Index unbenutzt zu lassen.

**Analyse:** Zur Ermittlung der Runs werden gegenüber reinem 2-Wege-Mergesort zusätzliche Schlüsselvergleiche ausgeführt, und zwar linear viele in jedem Durchgang. Bei $\lceil \log N \rceil$ Durchgängen im schlimmsten Fall ergeben sich damit zusätzlich $O(N \log N)$ Schlüsselvergleiche. Damit ist, wie auch schon beim reinen 2-Wege-Mergesort,

$$C_{max}(N) = \Theta(N \log N).$$

Der Vorzug des natürlichen 2-Wege-Mergesort liegt aber gerade in der Ausnutzung einer Vorsortierung. Im besten Fall ist die gegebene Folge bereits komplett sortiert, besteht also aus nur einem einzigen Run. Einmaliges Durchlaufen der Folge genügt um dies festzustellen und das Sortieren zu beenden. Also gilt

$$C_{min}(N) = \Theta(N).$$

Um die mittlere Anzahl $C_{mit}(N)$ von Schlüsselvergleichen zu bestimmen, überlegen wir uns, wie viele natürliche Runs eine zufällig gewählte Permutation von $N$ Schlüsseln im Mittel enthält. Betrachten wir zwei benachbarte Schlüssel $k_i$ und $k_{i+1}$ der zufällig gewählten Permutation. Dann ist die Wahrscheinlichkeit, dass $k_i < k_{i+1}$ ist, gleich der Wahrscheinlichkeit, dass $k_i > k_{i+1}$ ist, also gerade $1/2$ (unter der Annahme, dass alle Schlüssel verschieden sind). Falls $k_i > k_{i+1}$, dann ist bei $k_i$ ein Run zu Ende. Die Anzahl der Stellen, an denen ein Run zu Ende ist, ist also etwa $N/2$; damit ergeben sich im Mittel etwa $N/2$ Runs. Beim reinen 2-Wege-Mergesort erhalten wir bereits nach einem Durchlauf gerade $N/2$ Runs; daher sparen wir beim natürlichen 2-Wege-Mergesort lediglich einen Durchlauf im Mittel, also lediglich etwa $N$ Schlüsselvergleiche. Somit ergibt sich

$$C_{mit}(N) = \Theta(N \log N).$$

Anders ausgedrückt heißt das, dass im Mittel eine zufällig gewählte Schlüsselfolge nicht besonders gut vorsortiert ist, wenn man die Anzahl der Runs als Maß für die Vorsortierung wählt (vgl. dazu Abschnitt 2.6).

Die Anzahl der Bewegungen von Datensätzen lässt sich aufgrund dieser Überlegungen unmittelbar angeben:

$$M_{min}(N) = 0$$

und

$$M_{max}(N) = M_{mit}(N) = \Theta(N \log N).$$

Wenn Bewegungen von Datensätzen unerwünscht sind (z. B. wenn Datensätze groß sind), kann es vorteilhaft sein verkettete lineare Listen von Datensätzen zu sortieren. Dann genügt es nämlich die Zeiger von Listenelementen zu ändern; Bewegungen von Datensätzen erübrigen sich. Verkettete Listen sind deswegen für Mergesort-Varianten besonders geeignet, weil stets alle Teilfolgen nur sequenziell inspiziert werden; die Möglichkeit des Zugriffs auf beliebige Feldelemente haben wir nie in Anspruch genommen. Aus diesem Grund ist Mergesort auch ein gutes externes Sortierverfahren (vgl. Abschnitt 2.7).

Wir haben in allen Mergesort-Varianten die Prozedur *merge* für das Verschmelzen von zwei Teilfolgen verwendet. Dabei wurde linear viel zusätzlicher Speicherplatz benötigt. Es gibt auch Verfahren, die das Verschmelzen in situ, mit nur konstant viel zusätzlichem Speicherplatz bewerkstelligen und die trotzdem nur linear viele Schlüsselvergleiche ausführen (siehe z. B. [106] oder [200]).

# 2.5 Radixsort

In allen bisher behandelten Sortierverfahren waren Schlüsselvergleiche die einzige Informationsquelle um die richtige Anordnung der Datensätze zu ermöglichen. Wir haben zwar zur Vereinfachung stets vorausgesetzt, dass die Schlüssel ganzzahlig sind, die bisher besprochenen Sortierverfahren haben aber keine arithmetischen Eigenschaften der Schlüssel benutzt. Vielmehr wurde immer nur vorausgesetzt, dass das Universum der Schlüssel angeordnet ist und die relative Anordnung zweier Schlüssel durch einen in konstanter Zeit ausführbaren Schlüsselvergleich festgestellt werden kann.

Wir lassen diese Annahme jetzt fallen und nehmen an, dass die Schlüssel Wörter über einem aus $m$ Elementen bestehenden Alphabet sind. Beispiele sind:

$m = 10$ und die Schlüssel sind Dezimalzahlen;
$m = 2$ und die Schlüssel sind Dualzahlen;
$m = 26$ und die Schlüssel sind Wörter über dem Alphabet $\{a, \ldots, z\}$.

Man kann die Schlüssel also als $m$-adische Zahlen auffassen. Daher nennt man $m$ auch die Wurzel (lateinisch: *radix*) der Darstellung.

Für die in diesem Abschnitt besprochenen Sortierverfahren machen wir folgende, vereinfachende Annahme: Die Schlüssel der $N$ zu sortierenden Datensätze sind $m$-adische Zahlen gleicher Länge. Wenn alle Schlüssel verschieden sind, muss folglich die Länge wenigstens $\log_m N$ betragen. In Abschnitt 2.5.1 setzen wir sogar $m = 2$ voraus. Radix-Sortierverfahren inspizieren die einzelnen Ziffern der $m$-adischen Schlüssel. Wir setzen daher voraus, dass wir eine in konstanter Zeit ausführbare Funktion $z_m(i,k)$ haben, die für einen Schlüssel $k$ die Ziffer mit Gewicht $m^i$ in der $m$-adischen Darstellung von $k$, also die $i$-te Ziffer von rechts liefert, wenn man Ziffernpositionen ab 0 zu zählen beginnt. Es ist also z. B.:

$$z_{10}(0,517) = 7;$$
$$z_{10}(1,517) = 1;$$
$$z_{10}(2,517) = 5.$$

In Abschnitt 2.5.1 geben wir ein Radix-Sortierverfahren an, das eine rekursive Aufteilung des zu sortierenden Feldes analog zu Quicksort vornimmt. Dieses Verfahren hat in der Literatur den Namen *Radix-exchange-sort*. Das in Abschnitt 2.5.2 besprochene Radix-Sortierverfahren heißt Binsort, Bucketsort oder auch *Sortieren durch Fachverteilung*, weil es die zu sortierenden Datensätze wiederholt in Fächern (Bins, Buckets) ablegt, bis schließlich eine sortierte Reihenfolge vorliegt.

## 2.5.1 Radix-exchange-sort

**Methode:** Wir teilen das gegebene, nach aufsteigenden Binärschlüsseln gleicher Länge zu sortierende Feld $a[1] \ldots a[N]$ von Datensätzen in Abhängigkeit vom führenden Bit der binären Sortierschlüssel in zwei Teile. Alle Elemente, deren Schlüssel eine führende 0 haben, kommen in die linke Teilfolge und alle Elemente, deren Schlüssel eine

führende 1 haben, kommen in die rechte Teilfolge. Die Aufteilung wird ähnlich wie bei Quicksort in situ durch Vertauschen von Elementen des Feldes erreicht. Die Teilfolgen werden rekursiv auf dieselbe Weise sortiert, wobei natürlich jetzt das zweite Bit von links die Rolle des führenden Bits übernimmt.

Die Aufteilung eines Bereichs nach einer bestimmten Bitposition der Schlüssel des Bereichs erfolgt wie bei Quicksort. Man wandert mit zwei Zeigern vom linken und rechten Ende über den aufzuteilenden Bereich. Immer wenn der linke Zeiger auf ein Element stößt, das an der für die Aufteilung maßgeblichen Bitposition eine 1 hat, und der rechte auf ein Element stößt, das dort eine 0 hat, werden die beiden Elemente vertauscht. Die Aufteilung ist beendet, wenn die Zeiger übereinander gelaufen sind.

Abbildung 2.9 zeigt am Beispiel von sieben Binär-Schlüsseln der Länge 4 das Ergebnis der einzelnen Aufteilungsschritte.

| | | | Für die Aufteilung maßgebliches Bit: | | | |
|---|---|---|---|---|---|---|
| 1011 | 0010 | 1101 | 1001 | 0011 | 0101 | 1010 \| 3 |
| 0101 | 0010 | 0011 ; | 1001 | 1101 | 1011 | 1010 \| 2 |
| 0011 | 0010 ; | 0101 ; | 1001 | 1010 | 1011 ; | 1101 \| 1 |
| 0011 | 0010 ; | 0101 ; | 1001 ; | 1010 | 1011 ; | 1101 \| 0 |
| 0010 ; | 0011 ; | 0101 ; | 1001 ; | 1010 ; | 1011 ; | 1101 \| |

**Abbildung 2.9**

In dieser Tabelle ist das Ende der jeweils durch Aufteilung entstandenen Folgen durch „;“ markiert. Im Unterschied zu Quicksort kann man nicht direkt einen Stopper verwenden um das Wandern der Zeiger im Aufteilungsschritt zu beenden. Wir nehmen daher in die das Wandern der Zeiger kontrollierende Schleifenbedingung explizit den Test auf, ob ein Zeiger das Ende des jeweils aufzuteilenden Bereichs bereits erreicht hat.

```
procedure radixexchangesort (var a : sequence; l, r, b : integer);
  {sortiert die Elemente a[l]...a[r] nach aufsteigenden Werten
   der Endstücke von Schlüsseln, die aus Bits
   an den Positionen 0,...,b bestehen}
  var
    i, j : integer; {Zeiger}
    t : item; {Hilfsspeicher}
  begin
    if r > l
      then
        begin
          {teile Bereich a[l]...a[r] abhängig vom Bit an Position b
           der Schlüssel auf}
          i := l − 1;
          j := r + 1;
```

```
            begin-loop
              repeat i := i+1 until (z2(b,a[i].key) = 1) or (i ≥ j);
              repeat j := j-1 until (z2(b,a[j].key) = 0) or (i ≥ j);
              if i ≥ j
                then exit-loop;
              t := a[i];
              a[i] := a[j];
              a[j] := t
            end-loop;
            {alle Elemente mit einer 0 an Bit-Position b stehen jetzt
             links von allen Elementen mit einer 1 an dieser Position;
             i zeigt auf den Beginn dieser rechten Teilfolge.
             Gibt es keine Elemente in a[l]...a[r]  mit einer 1
             an Bitposition b, so ist i = r+1}
            if b > 0  {das 0-te Bit ist noch nicht inspiziert}
              then
                begin
                  radixexchangesort(a,l,i-1,b-1);
                  radixexchangesort(a,i,r,b-1)
                end
        end
    end
```

Ein Aufruf von *radixexchangesort*$(a,1,N,$*Schlüssellänge*$)$ sortiert dann das gegebene Feld.

Der Aufteilungsschritt für einen Bereich $a[l]\ldots a[r]$, abhängig vom Schlüsselbit an Position $b$, benötigt wie bei Quicksort $c\cdot(r-l)$ Schritte mit einer Konstanten $c$. Zwar kann die Prozedur *radixexchangesort* für Schlüssel der Länge $b+1$ insgesamt $(2^{b+1}-1)$-mal rekursiv aufgerufen werden, aber die maximale Rekursionstiefe ist höchstens $b$. Alle Aufteilungen auf derselben Rekursionsstufe, also alle von derselben Bitposition abhängigen Aufteilungen, können insgesamt in linearer Zeit ausgeführt werden. Daraus folgt, dass die Laufzeit des Verfahrens – unabhängig von der Eingabefolge – stets durch $O(N\cdot b)$ abgeschätzt werden kann. Ist $b=\log N$, dann ist Radix-exchange-sort eine echte Alternative zu Quicksort. Hat man aber wenige lange Schlüssel, ist Radix-exchange-sort schlecht.

### 2.5.2 Sortieren durch Fachverteilung

In der Anfangszeit der Datenverarbeitung gab es mechanische Geräte zum Sortieren eines Lochkartenstapel. Der zu sortierende Kartenstapel wird (in Abhängigkeit von der Lochung an einer bestimmten Position) auf verschiedene Fächer verteilt. Die in den Fächern abgelegten Teilstapel werden dann in einer bestimmten, festen Reihenfolge eingesammelt und erneut, abhängig von der nächsten Lochkartenposition, verteilt usw., bis schließlich ein sortierter Stapel entstanden ist. Charakteristisch für dieses Sortierverfahren ist also der Wechsel zwischen einer *Verteilungsphase* und einer *Sammelphase*. Wir beschreiben beide Phasen nun genauer und setzen dazu voraus, dass das Feld

der zu sortierenden Datensätze $a[1]\ldots a[N]$ $m$-adische Schlüssel gleicher Länge $l$ hat. Verteilungs- und Sammelphase werden insgesamt $l$-mal durchgeführt. Denn die Verteilungsphase hängt ab von der jeweils gerade betrachteten Ziffer an Position $t$ der $m$-adischen Schlüssel, wobei $t$ die Positionen von 0 bis $l-1$ durchläuft, also von der niedrigst wertigen zur höchst wertigen Ziffernposition.

In der *Verteilungsphase* werden die Datensätze auf $m$ Fächer verteilt. Das $i$-te Fach $F_i$ nimmt alle Datensätze auf, deren Schlüssel an Position $t$ die Ziffer $i$ haben. Der jeweils nächste Satz wird stets „oben“ auf die in seinem Fach bereits vorhandenen Sätze gelegt.

In der *Sammelphase* werden die Sätze in den Fächern $F_0,\ldots,F_{m-1}$ so eingesammelt, dass die Sätze im Fach $F_{i+1}$ als Ganzes „oben“ auf die Sätze im Fach $F_i$ gelegt werden (für $0 \le i < m-1$). Die relative Anordnung der Sätze innerhalb eines jeden Fachs bleibt unverändert.

In der auf eine Sammelphase folgenden Verteilungsphase müssen die Datensätze von „unten“ nach „oben“ verteilt werden, also zuerst wird der „unterste“ Satz in sein Fach gelegt, dann der „zweitunterste“ usw., bis schließlich der „oberste“ Satz auf ein Fach verteilt ist. Am Ende der letzten Sammelphase sind dann die Datensätze von „unten“ nach „oben“ sortiert.

Wir illustrieren das Verfahren am Beispiel einer Menge von 12 Datensätzen mit zweistelligen Dezimalschlüsseln. (D. h. wir haben $N=12$, $m=10$, $l=2$.) Gegeben sei die unsortierte Schlüsselfolge

$$40,\ 13,\ 22,\ 54,\ 15,\ 28,\ 76,\ 04,\ 77,\ 38,\ 16,\ 18.$$

Während der ersten Verteilungsphase werden die Schlüssel in Abhängigkeit von der am weitesten rechts stehenden Dezimalziffer (an Position $t=0$) wie folgt auf zehn Fächer verteilt.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | 18 | |
| | | | | 04 | | 16 | | 38 | |
| 40 | | 22 | 13 | 54 | 15 | 76 | 77 | 28 | |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ |

Nach der ersten Sammelphase ergibt sich die Schlüsselfolge

$$40,\ 22,\ 13,\ 54,\ 04,\ 15,\ 76,\ 16,\ 77,\ 28,\ 38,\ 18.$$

Erneute Verteilung nach der Ziffer an Position $t=1$ ergibt:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 18 | | | | | | | | |
| | 16 | | | | | | | | |
| | 15 | 28 | | | | | 77 | | |
| 04 | 13 | 22 | 38 | 40 | 54 | | 76 | | |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ |

Sammeln der Schlüssel in den Fächern ergibt die sortierte Schlüsselfolge

$$04,\ 13,\ 15,\ 16,\ 18,\ 22,\ 28,\ 38,\ 40,\ 54,\ 76,\ 77.$$

Für den Nachweis der Korrektheit des Verfahrens ist die folgende Beobachtung wichtig: Die von der Ziffer an Position $t$ abhängige Verteilungs- und Sammelphase liefert eine

aufsteigende Sortierung der Sätze nach dieser Schlüsselziffer; dabei bleibt die relative Anordnung der Schlüssel innerhalb der Fächer erhalten. Genauer: War die Zahlenfolge vor Beginn eines Durchgangs (bestehend aus Verteilungs- und Sammelphase) nach aufsteigenden Werten sortiert, wenn man nur die Ziffernpositionen $0,\ldots,t-1$ betrachtet (bzw. unsortiert, wenn $t=0$ ist), so folgt: Nach Ende des Durchgangs sind die Schlüssel bzgl. der Ziffernpositionen $0,\ldots,t$ aufsteigend sortiert. $l$ Durchgänge stellen also eine insgesamt sortierte Reihenfolge her, wenn dabei der Reihe nach die Ziffernpositionen $0,\ldots,l-1$ betrachtet werden.

Eine naive Implementation des Verfahrens erfordert die programmtechnische Realisierung von $m$ Fächern als Felder der Größe $N$; denn jedes Fach kann ja im ungünstigsten Fall alle $N$ Datensätze aufnehmen müssen. Das ist schon für $m=10$ kein gangbarer Weg, weil dann zur Sortierung von $N$ Datensätzen $m\cdot N$ zusätzliche Speicherplätze reserviert werden müssen. Es gibt zwei nahe liegende Möglichkeiten diese enorme Speicherplatzverschwendung zu vermeiden. Eine erste Möglichkeit ist zu Beginn eines jeden Durchlaufs (dessen Verteilungsphase von der $t$-ten Ziffer abhängt) zu zählen wie viele Sätze in jedes Fach fallen werden. Da insgesamt nicht mehr als $N$ Datensätze verteilt werden müssen, genügt es dann, insgesamt $N$ zusätzliche Speicherplätze zu vereinbaren. Man kann in diesem Bereich alle Fächer unterbringen und die jeweiligen Bereiche der einzelnen Fächer aus den zuvor ermittelten Anzahlen leicht bestimmen. Wir nennen die Indizes der Grenzen dieser Bereiche die *Verteilungszahlen*. Eine andere Möglichkeit ist die $m$ Fächer als $m$ verkettete lineare Listen mit variabler Länge zu realisieren. In der Verteilungsphase werden die Datensätze stets an das Ende der jeweiligen Liste angehängt. In der Sammelphase werden die Listen der Reihe nach von vorn nach hinten durchlaufen.

Für beide Varianten werden Prozeduren angegeben, die eine Sortierung des Feldes $a$ vom Typ *sequence* mit ganzzahligen Schlüsselkomponenten $a[i].key$ vornehmen. Wir nehmen an, dass die Schlüssel $m$-adische Zahlen der Länge $l$ sind, wobei $m$ und $l$ außerhalb der Prozeduren festgelegte Konstanten sind. Wir erinnern daran, dass für jede Zahl $t$, mit $0\le t<m$, $z_m(t,a[i].key)$ die $t$-te Ziffer des $m$-adischen Schlüssels des $i$-ten Satzes ist.

```
procedure radixsort_1 (var a : sequence);
  var
    b : sequence; {Speicher zur Aufnahme der Fächer}
    c : array [0 .. m] of integer; {Verteilungszahlen}
    i, j, t : integer;
  begin
    for t := 0 to l − 1 do
      begin {Durchlauf}
        {Verteilungsphase: Verteilungszahlen bestimmen}
        for i := 0 to m − 1 do c[i] := 0;
        for i := 1 to N do
          begin
            j := z_m(t, a[i].key);
            c[j] := c[j] + 1
          end;
        c[m − 1] := N + 1 − c[m − 1];
```

```
        for i := 2 to m do c[m − i] := c[m − i + 1] − c[m − i];
        {c[i] ist Index des Anfangs von Fach F_i im Feld b}
        {verteilen}
        for i := 1 to N do
          begin
            j := z_m(t, a[i].key);
            b[c[j]] := a[i];
            c[j] := c[j] + 1
          end;
        {Sammelphase}
        for i := 1 to N do a[i] := b[i]
      end {Durchlauf}
  end
```

Für die zweite Radixsort-Variante machen wir Gebrauch von dem in Kapitel 1 beschriebenen Datentyp Liste, der die Menge aller Listen von Objekten eines gegebenen Grundtyps einschließlich der leeren Liste bezeichnet und mit **list of** vereinbart wird. Wir erinnern daran, dass für eine Variable $L$ vom Typ Liste, also

**var** $L$ : **list of** *Grundtyp*

und eine Variable $x$ des Grundtyps, also

**var** $x$ : *Grundtyp*

die folgenden Prozeduren und Funktionen für $L$ und $x$ erklärt und in konstanter Zeit ausführbar sind:

| | |
|---|---|
| $pushtail(L,x)$ : | hängt $x$ an das Ende von $L$ an; das Resultat ist $L$; |
| $pophead(L,x)$ : | entfernt das erste Element aus $L$; die entstehende Liste ist $L$; das entfernte Element ist $x$; |
| $empty(L)$ : | liefert den Wert *true* genau dann, wenn $L$ die leere Liste ist, und den Wert *false* sonst; |
| $init(L)$ : | liefert für $L$ die leere Liste. |

Dann lässt sich die Prozedur *radixsort_2* wie folgt angeben:

```
procedure radixsort_2 (var a : sequence);
var
  L : array [0 .. (m − 1)] of list of item;
  i, j, t : integer;
begin
  for j := 0 to (m − 1) do init(L[j]); {Fächer leeren}
  for t := 0 to l − 1 do
    begin {Durchlauf}
      {Verteilungsphase}
      for i := 1 to N do {verteilen}
        begin
          j := z_m(t, a[i].key);
```

```
                pushtail(L[j], a[i])
              end;
            {Sammelphase}
            i := 1;
            for j := 0 to m − 1 do {L[j] einsammeln}
              while not empty(L[j]) do
                begin
                  pophead(L[j], a[i]);
                  i := i + 1
                end {while}
        end {Durchlauf}
    end
```

Aus den angegebenen Programmstücken kann man unmittelbar ablesen, dass beide Radixsort-Versionen in $O(l(m+N))$ Schritten ausführbar sind. Der Speicherbedarf liegt in beiden Fällen in der Größenordnung $O(N+m)$.

Wir diskutieren einige Spezialfälle genauer. Sollen $m$ verschiedene Schlüssel im Bereich $0,\ldots,m-1$ sortiert werden, so ist also $m = N$ und $l = 1$. In diesem Fall liefert Radixsort eine sortierte Folge in linearer Zeit und mit linearem Platz. Dieser sehr spezielle Fall kann natürlich viel einfacher wie folgt gelöst werden. Man vereinbart ein Feld $b$ vom Typ

$$\textbf{array}[0\,..\,(m-1)]\ \textbf{of}\ item$$

und erreicht durch die Anweisung

$$\textbf{for}\ i := 1\ \textbf{to}\ N\ \textbf{do}\ b[a[i].key] := a[i],$$

dass die Sätze des gegebenen Feldes $a$ in $b$ nach aufsteigenden Schlüsseln sortiert vorkommen. Die Sortierung ist hier also trivial erreichbar, weil jedes „Fach“ genau einen Satz aufnimmt.

Haben die gegebenen $N$ Datensätze Schlüssel fester Länge $l$ im Bereich $0,\ldots,m^l-1$, ist also $l$ konstant und $m < N$, so ist Radixsort in linearer Zeit ausführbar. Es ist klar, dass $l \geq \lceil \log_m N \rceil$ sein muss, wenn alle $N$ Schlüssel verschieden sind. Solange die Schlüssel „kurze“ $m$-adische Zahlen sind, also $l = c \cdot \lceil \log_m N \rceil$ mit einer „kleinen“ Konstanten $c$, bleibt Radixsort ein praktisch brauchbares Verfahren mit einer Gesamtlaufzeit von $O(N \log N)$ im schlechtesten Fall.

## 2.6 Sortieren vorsortierter Daten

Nicht selten sind zu sortierende Datenbestände bereits teilweise vorsortiert. Sie können etwa aus einigen bereits sortierten Teilen zusammengesetzt sein oder an ein größeres, sortiertes File werden am Ende einige wenige Sätze in beliebiger Reihenfolge angehängt. Viele Sortierverfahren ziehen aber aus einer Vorsortierung keinerlei Nutzen.

Schlimmer noch: Manche Sortierverfahren, wie etwa Quicksort, sind für vorsortierte Daten sogar besonders schlecht. Damit stellt sich die Frage: Gibt es Sortierverfahren, die die in einem Datenbestand bereits vorhandene Vorsortierung optimal nutzen?

In dieser Form ist die Frage natürlich viel zu unpräzise formuliert um eine klare ja/nein Antwort zu erlauben. Wir werden daher in Abschnitt 2.6.1 zunächst einige gebräuchliche Maße zur Messung der Vorsortierung einer Folge von Schlüsseln vorstellen und präzise definieren, was es heißt, dass ein Sortierverfahren von einer so gemessenen Vorsortierung optimalen Gebrauch macht. In Abschnitt 2.6.2 stellen wir ein erstes „adaptives“ Sortierverfahren vor, das die Vorsortierung einer Folge optimal nutzt, wenn man sie mit der Inversionszahl misst. Das in Abschnitt 2.6.3 besprochene *Sortieren durch lokales Einfügen* ist sogar für drei verschiedene Vorsortierungsmaße optimal.

## 2.6.1 Maße für Vorsortierung

Betrachten wir einige verschiedene Folgen von 9 Schlüsseln, die aufsteigend sortiert werden sollen:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $F_a$ : | 2 | 1 | 4 | 3 | 6 | 5 | 8 | 7 | 9 |
| $F_b$ : | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 |
| $F_c$ : | 5 | 1 | 7 | 4 | 9 | 2 | 8 | 3 | 6 |

Intuitiv würde man sagen: Die Folge $F_c$ ist weniger vorsortiert als $F_a$ und $F_b$. $F_a$ ist global schon ganz gut sortiert, denn kleine Schlüssel stehen eher am Anfang, große Schlüssel eher am Ende der Folge. Die Unordnung in $F_a$ ist also lokaler Natur. Tatsächlich sind gegenüber der sortierten Folge einfach Paare benachbarter Schlüssel vertauscht. Das umgekehrte gilt für Folge $F_b$. Lokal ist $F_b$ ganz gut sortiert, denn die meisten Paare benachbarter Schlüssel in $F_b$ stehen in der richtigen Reihenfolge, aber global ist $F_b$ ziemlich ungeordnet, denn große Schlüssel stehen am Anfang, kleine am Ende der Folge. Wie lässt sich das quantitativ messen? Wir machen dazu jetzt drei verschiedene Vorschläge und diskutieren ihre Vor- und Nachteile.

Die erste Möglichkeit besteht darin, die Anzahl der *Inversionen* (oder: Fehlstellungen) zu messen. Das ist die Anzahl von Paaren von Schlüsseln, die in der falschen Reihenfolge stehen. In der Beispielfolge $F_a$ sind dies die vier Paare (2,1), (4,3), (6,5), (8,7) und in der Beispielfolge $F_b$ die 20 Paare

(6,1), (6,2), (6,3), (6,4), (6,5),
(7,1), (7,2), (7,3), (7,4), (7,5),
(8,1), (8,2), (8,3), (8,4), (8,5),
(9,1), (9,2), (9,3), (9,4), (9,5).

Allgemein: Sei $F = \langle k_1, \ldots, k_N \rangle$ eine Folge von Schlüsseln, die aufsteigend sortiert werden soll. Wir setzen voraus, dass alle Schlüssel $k_i$ verschiedene (ganze) Zahlen sind. Dann heißt die Anzahl der Paare in falscher Reihenfolge

$$inv(F) = \left|\left\{(i,j) | 1 \leq i < j \leq N \text{ und } k_i > k_j\right\}\right|$$

die *Inversionszahl* von $F$. Falls $F$ bereits aufsteigend sortiert ist, so ist offenbar $inv(F) = 0$. Im ungünstigsten Fall kann jedes Element $k_i$ in einer Folge $F$ vor Elementen $k_{i+1}, \ldots, k_N$ stehen, die sämtlich kleiner als $k_i$ sind. Dies ist der Fall für eine absteigend sortierte Folge $F$, für deren Inversionszahl offenbar gilt:

$$inv(F) = (N-1) + (N-2) + \cdots + 2 + 1 = \frac{N(N-1)}{2}.$$

Die Inversionszahl misst die globale Vorsortierung. Die oben angegebenen Beispielfolgen zeigen das sehr deutlich: Die Folge $F_a$ hat eine kleine, die Folge $F_b$ hat eine große Inversionszahl. Dennoch würde man auch Folge $F_b$ als gut vorsortiert ansehen; sie lässt sich leicht und schnell etwa mit dem Verfahren Sortieren durch Verschmelzen in eine sortierte Reihenfolge bringen. Das nächste Maß berücksichtigt diese Art Vorsortierung besser. Man nimmt die Anzahl der bereits aufsteigend sortierten Teilfolgen einer gegebenen Folge. Diese Zahl heißt die *Run-Zahl* (englisch: run = Lauf). Sie ist für eine Folge $F = \langle k_1, \ldots, k_N \rangle$ wie folgt definiert:

$$runs(F) = |\{i \mid 1 \leq i < N \text{ und } k_{i+1} < k_i\}| + 1.$$

Für die Folge $F_b$ ist offenbar $runs(F_b) = 2$, weil in $F_b$ nur eine Stelle vorkommt, an der ein größeres Element einem kleineren unmittelbar vorangeht. In der Folge $F_a$ gibt es vier solcher Stellen, die wir mit einem „↑“ markiert haben.

$$\begin{array}{ccccccccc} 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 & 9 \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & \end{array}$$

Also ist $runs(F_a) = 4 + 1 = 5$.

Für eine bereits aufsteigend sortierte Folge ist die Run-Zahl 1. Im ungünstigsten Fall kann an jeder Stelle zwischen je zwei benachbarten Elementen ein größeres einem kleineren Folgenelement unmittelbar vorangehen. Das ist der Fall für absteigend sortierte Folgen. Die Run-Zahl einer absteigend sortierten Folge der Länge $N$ ist also $N$. Eine kleine Run-Zahl ist also ein Indiz für einen hohen Grad von Vorsortiertheit. Die Run-Zahl ist aber eher ein lokales Maß für die Vorsortierung. Denn eine intuitiv gut vorsortierte Folge mit nur lokaler Unordnung, wie die Folge $\langle 2,1,4,3,\ldots,N,N-1 \rangle$ hat eine hohe Run-Zahl (von ungefähr $N/2$).

Ein die genannten Nachteile (vorwiegend lokal oder vorwiegend global orientiert) vermeidendes Maß für die Vorsortierung beruht auf der Messung der *längsten aufsteigenden Teilfolge las* (longest ascending subsequence) einer Folge $F = \langle k_1, \ldots, k_N \rangle$:

$$\begin{aligned} las(F) \quad = \quad & \max\{t \mid \exists i(1), \ldots, i(t) \text{ so dass} \\ & 1 \leq i(1) < \ldots < i(t) \leq N \text{ und } k_{i(1)} < \ldots < k_{i(t)}\}. \end{aligned}$$

Offensichtlich gilt für eine Folge $F$ der Länge $N$ stets $1 \leq las(F) \leq N$. Für eine gut vorsortierte Folge $F$ ist $las(F)$ groß und für eine schlecht vorsortierte, wie die absteigend sortierte Folge $F$, ist $las(F)$ klein. $las(F)$ wächst also gerade umgekehrt wie die beiden vorher eingeführten Maße *inv* und *runs*. Man benutzt daher besser die Differenz $N - las(F)$ zur Messung der Vorsortierung. Sie gibt an, wie viele Elemente von $F$

wenigstens entfernt werden müssen (englisch: remove), um eine aufsteigend sortierte Folge zu hinterlassen:

$$rem(F) = N - las(F).$$

Die oben angegebene Beispielfolge $F_a$ hat mehrere längste aufsteigende Teilfolgen mit Länge fünf. $F_b$ hat eine längste aufsteigende Teilfolge ebenfalls mit Länge fünf. Also gilt für beide Folgen

$$rem(F_a) = rem(F_b) = 9 - 5 = 4.$$

Das Maß *rem* ist weniger intuitiv als die Maße *inv* und *runs*. Die Berechnung von $rem(F)$ für eine gegebene Folge $F$ ist ein durchaus nicht triviales algorithmisches Problem. Wir verweisen hier nur auf die Arbeit von H. Mannila [128].

Es gibt in der Literatur noch weitere Vorschläge zur Messung der Vorsortiertheit von Schlüsselfolgen und auch den Versuch einer allgemeinen Theorie durch axiomatische Fassung der ein Maß für Vorsortierung charakterisierenden Eigenschaften. Da es uns nur auf einige grundsätzliche Aspekte des Problems, vorsortierte Folgen möglichst effizient zu sortieren, ankommt, wollen wir uns mit der Betrachtung der drei oben angegebenen Maße begnügen.

Wir wollen jetzt präzisieren, was es heißt, dass ein Sortierverfahren Vorsortierung optimal nutzt, wenn man ein Maß $m$ zur Messung der Vorsortierung wählt. Erinnern wir uns daran, dass jeder Algorithmus zur Lösung des Sortierproblems zwei Teilprobleme löst, und zwar ein Informationsbeschaffungsproblem und ein Datentransportproblem. Die sortierte Schlüsselfolge muss zunächst unter allen anderen Folgen (gleicher Länge) identifiziert werden. Im einfachsten Fall geschieht dies durch Ausführen von Vergleichsoperationen zwischen je zwei Schlüsseln. Algorithmen, die nur auf diese Weise Informationen über die (relative) Anordnung der zu sortierenden Schlüssel gewinnen, heißen *allgemeine* Sortierverfahren. Sämtliche bisher besprochenen Verfahren mit Ausnahme von Radixsort gehören zu dieser Klasse. Zweitens ist ein Datentransportproblem zu lösen. Die zu sortierende Folge muss durch Bewegen von Datensätzen (oder Zeigern) in die richtige Reihenfolge gebracht werden.

Für die Definition eines *m-optimalen Sortierverfahrens*, wobei $m$ ein Maß für die Vorsortierung ist, beschränken wir uns auf die Klasse der allgemeinen Sortierverfahren. Dann kann man als untere Schranke für die Laufzeit eines solchen Verfahrens die zum Sortieren ausgeführte Zahl von Schlüsselvergleichen nehmen.

Wann ist ein Sortierverfahren $m$-optimal? Intuitiv doch dann, wenn das Verfahren zum Sortieren einer Folge $F$ auch nur die für Folgen dieses Vorsortiertheitsgrades minimal nötige Schrittzahl tatsächlich benötigt. Folgen mit kleinem $m(F)$-Wert sollen schnell, Folgen mit größerem $m(F)$-Wert entsprechend langsamer sortiert werden. Versuchen wir zunächst, die mindestens erforderliche Anzahl von Schlüsselvergleichen abzuschätzen für ein gegebenes Sortierverfahren. Wir nehmen an, dass das Verfahren keine überflüssigen Schlüsselvergleiche ausführt.

Wir können das Verfahren wie jedes allgemeine Sortierverfahren durch einen so genannten *Entscheidungsbaum* mit genau $N!$ Blättern repräsentieren (vgl. auch Abschnitt 2.8). Der Entscheidungsbaum ist ein Mittel zur statischen Veranschaulichung aller Schlüsselvergleiche, die zum Sortieren aller $N!$ Folgen der Länge $N$ mithilfe eines gegebenen Verfahrens ausgeführt werden. Jedem Pfad von der Wurzel zu einem Blatt in diesem Baum entspricht die zur Identifizierung (und Sortierung) einer bestimmten Schlüsselfolge ausgeführte Folge von Vergleichsoperationen.

Wir betrachten als Beispiel in Abbildung 2.10 einen Entscheidungsbaum für vier verschiedene Schlüssel.

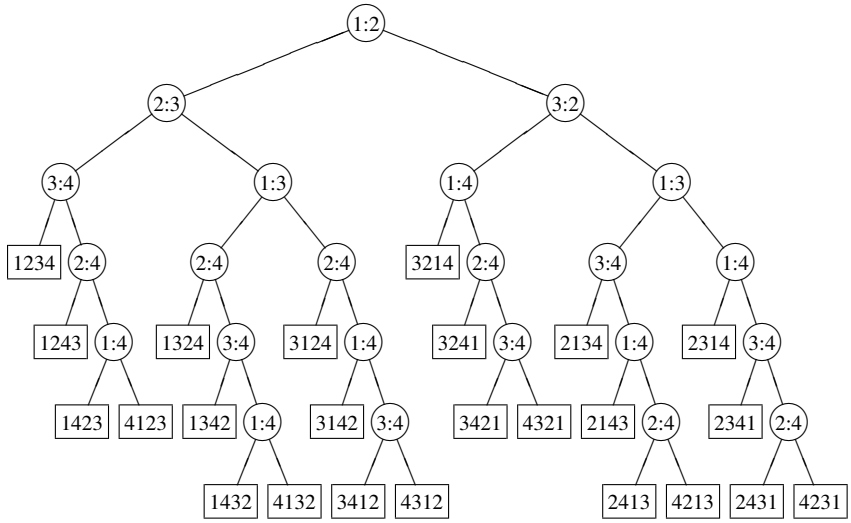



**Abbildung 2.10**

Er modelliert das Verhalten eines bestimmten allgemeinen Sortierverfahrens für Schlüsselfolgen der Länge 4. Ist eine Folge $F = \langle k_1, k_2, k_3, k_4 \rangle$ gegeben, so werden zunächst die Schlüssel $k_1$ und $k_2$ miteinander verglichen. Ist $k_1 < k_2$, werden als nächste die Schlüssel $k_2$ und $k_3$ miteinander verglichen; das entspricht dem Hinabsteigen von dem mit $1:2$ beschrifteten Wurzelknoten im Entscheidungsbaum zu dessen linkem Sohn. Falls der Vergleich zwischen $k_1$ und $k_2$ ergeben hätte, dass $k_1 > k_2$, würde dann anschließend $k_3$ mit $k_2$ verglichen und dann $k_1$ mit $k_4$ bzw. $k_1$ mit $k_3$, je nachdem, ob $k_3 < k_2$ war oder nicht.

Jedem Pfad von der Wurzel des Entscheidungsbaumes zu einem Blatt entspricht also diejenige Folge von Schlüsselvergleichen, die zur Identifizierung und Sortierung der durch das Blatt repräsentierten Eingabefolge ausgeführt wird. Beispielsweise repräsentiert das Blatt $\boxed{3\ 2\ 1\ 4}$ die aufsteigend sortierte Schlüsselfolge, für die $k_3 < k_2 < k_1 < k_4$ gilt. Um sie zu identifizieren, werden nacheinander $k_1$ mit $k_2$, $k_3$ mit $k_2$ und $k_1$ mit $k_4$ verglichen, d. h. nacheinander die Knoten $1:2$, $3:2$ und $1:4$ betrachtet. Zur Identifizierung dieser Folge werden also nur wenige Schlüsselvergleiche ausgeführt.

Ein allgemeines Sortierverfahren $A$ nutzt die mit einem Maß $m$ gemessene Vorsortierung voll aus, wenn alle bezüglich $m$ maximal vorsortierten Folgen im $A$ entsprechenden Entscheidungsbaum durch Blätter repräsentiert werden, die „so nah wie möglich“ bei der Wurzel sind. Man betrachtet für eine gegebene Folge $F$ also die Menge aller Folgen $F'$ gleicher Länge, die genau so gut wie $F$ oder besser vorsortiert sind. Sei $r$ ihre Anzahl, also:

$$r = \left|\left\{F' \mid m(F') \leq m(F)\right\}\right|$$

Weil Entscheidungsbäume Binärbäume sind, die auf jedem Niveau $i$ höchstens $2^i$ Blätter haben können, folgt: Es gibt wenigstens eine Folge $F_0 \in \{F' \mid m(F') \leq m(F)\}$, deren Abstand von der Wurzel des Entscheidungsbaumes wenigstens $\lceil \log r \rceil$ ist. Darüberhinaus muss auch der mittlere Abstand aller Blätter, die Folgen in der Menge $\{F' \mid m(F') \leq m(F)\}$ entsprechen, in $\Omega(\log r)$ sein. (Für einen Beweis dieser Tatsache vgl. Abschnitt 2.8.)

Anders formuliert: Jeder Algorithmus muss zur Identifizierung (und Sortierung) einer Menge von Folgen gleicher Länge und mit gegebenem Vorsortierungsgrad $g$ sowohl im Mittel wie im schlechtesten Fall wenigstens

$$\Omega\left(\log\left(\left|\left\{F' \mid m(F') \leq g\right\}\right|\right)\right)$$

Vergleichsoperationen zwischen Schlüsseln ausführen.

Man wird ein Verfahren sicher dann *m-optimal* nennen, wenn es diese untere Schranke bis auf einen konstanten Faktor erreicht. Diese Forderung ist allerdings etwas zu scharf, weil die Menge aller Folgen der Länge $N$ mit einem gegebenen Vorsortierungsgrad weniger als $2^N$ Elemente haben kann. Ein konstantes Vielfaches der Mindestschrittzahl würde dann eine sublineare Laufzeit verlangen. Man erwartet aber andererseits, dass jedes Sortierverfahren jedes Element wenigstens einmal betrachten muss und damit wenigstens Zeitbedarf $\Omega(N)$ hat.

Damit haben wir die endgültige Definition eines *m-optimalen Sortierverfahrens A*. $A$ heißt $m$-optimal, wenn es eine Konstante $c$ gibt, sodass für alle $N$ und alle Folgen $F$ mit Länge $N$ gilt: $A$ sortiert $F$ in Zeit

$$T_A(F,m) \leq c \cdot \left(N + \log\left(\left|\left\{F' \mid m(F') \leq m(F)\right\}\right|\right)\right).$$

## 2.6.2 A-sort

Wir wollen jetzt Sortierverfahren angeben, die Vorsortierung im zuvor präzisierten Sinne optimal nutzen. Wir beginnen mit dem Vorsortierungsmaß *inv* und fragen uns, nach welcher Strategie ein Sortierverfahren arbeiten muss, das die mit der Inversionszahl gemessene Vorsortierung optimal ausnutzt.

Für eine Folge $F = \langle k_1, \ldots, k_N \rangle$ von Schlüsseln ist offenbar

$$inv(F) = \left|\left\{(i,j) \mid 1 \leq i < j \leq N \text{ und } k_i > k_j\right\}\right| \quad = \quad \sum_{j=1}^{N} h_j$$

mit

$$h_j = \left|\left\{i \mid 1 \leq i < j \leq N \text{ und } k_i > k_j\right\}\right|.$$

Für jedes $j$ ist $h_j$ die Anzahl der dem $j$-ten Element $k_j$ in der gegebenen Folge vorangehenden Elemente, die bei aufsteigender Sortierung $k_j$ nachfolgen müssen. Die Größen $h_j$ lassen sich also auch so deuten: Fügt man $k_1, k_2, \ldots$ der Reihe nach in die anfangs leere und sonst stets aufsteigend sortierte Liste ein, so gibt $h_j$ den Abstand des jeweils nächsten einzufügenden Elementes $k_j$ vom Ende der bisher erhaltenen Liste an, die bereits die Elemente $k_1, \ldots, k_{j-1}$ enthält.

Betrachten wir ein Beispiel (Folge $F_c$ aus Abschnitt 2.6.1):

$$F = \langle k_1, \ldots, k_9 \rangle = \langle 5,1,7,4,9,2,8,3,6 \rangle$$

Für diese Folge ergeben sich die in Tabelle 2.3 gezeigten Einzelschritte.

| nächstes einzufügendes Element $k_i$ | $h_i$ = Abstand der Einfügestelle vom Ende der bisherigen Liste | nach Einfügen erhaltene Liste mit Markierung der Einfügestelle |
|---|---|---|
| $k_1 = 5$ | $h_1 = 0$ | $\underline{5}$ |
| $k_2 = 1$ | $h_2 = 1$ | $\underline{1}$, 5 |
| $k_3 = 7$ | $h_3 = 0$ | 1, 5, $\underline{7}$ |
| $k_4 = 4$ | $h_4 = 2$ | 1, $\underline{4}$, 5, 7 |
| $k_5 = 9$ | $h_5 = 0$ | 1, 4, 5, 7, $\underline{9}$ |
| $k_6 = 2$ | $h_6 = 4$ | 1, $\underline{2}$, 4, 5, 7, 9 |
| $k_7 = 8$ | $h_7 = 1$ | 1, 2, 4, 5, 7, $\underline{8}$, 9 |
| $k_8 = 3$ | $h_8 = 5$ | 1, 2, $\underline{3}$, 4, 5, 7, 8, 9 |
| $k_9 = 6$ | $h_9 = 3$ | 1, 2, 3, 4, 5, $\underline{6}$, 7, 8, 9 |

**Tabelle 2.3**

Ist die $\sum h_j$, also die Inversionszahl, klein, so müssen auch die $h_j$ (im Durchschnitt) klein sein; d. h. das jeweils nächste Element wird nah am rechten Ende der bisher erzeugten Liste eingefügt. Im Extremfall einer bereits aufsteigend sortierten Folge mit Inversionszahl 0 und $h_1 = \ldots = h_N = 0$ wird jedes Element ganz am rechten Ende eingefügt. Um Folgen mit kleiner Inversionszahl schnell zu sortieren, sollte man also der *Strategie des Sortierens durch iteriertes Einfügen* folgen und dabei eine Struktur „dynamische, sortierte Liste“ verwenden, die das Einfügen in der Nähe des Listenendes effizient erlaubt. Ein Array lässt sich dafür nicht nehmen. Denn man kann unter Umständen zwar die Einfügestelle für das nächste Element schnell finden (etwa mit binärer oder exponentieller Suche, vgl. hierzu das Kapitel 3), muss aber viele Elemente verschieben um das nächste Element an der richtigen Stelle unterzubringen. Eine verkettete lineare, aufsteigend sortierte Liste mit einem Zeiger auf das Listenende, die vom Ende her durchsucht (und vom Anfang an ausgegeben) werden kann, ist ebenfalls nicht besonders gut. Zwar kann man in eine derartige Struktur ein neues Element in konstanter Schrittzahl einfügen, sobald man die Einfügestelle gefunden hat. Zum Finden der Einfügestelle benötigt man aber $h$ Schritte, wenn sie den Abstand $h$ vom Listenende hat. Was man brauchen könnte, ist eine Struktur, die beide Vorteile – schnelles Finden der Einfügestelle nahe dem Listenende und schnelles Einfügen – miteinander verbindet.

Strukturen zur Speicherung sortierter Folgen, die das Suchen und Einfügen eines neuen Elementes im Abstand $h$ vom Ende der Folge in $O(\log h)$ Schritten erlauben, gibt es in der Tat. Geeignet gewählte Varianten balancierter, blattorientierter Suchbäume haben die verlangten angenehmen Eigenschaften, wenn man Suchen und Einfügen richtig implementiert. Wir skizzieren hier grob die der Lösung zu Grunde liegende Idee und verweisen auf das Kapitel über Bäume für die Details.

Die sortierte Folge wird in einer aufsteigend sortierten, verketteten Liste gespeichert. Ein üblicherweise *Finger* genannter Zeiger weist auf das Listenende mit dem jeweils größten Element. Über dieser Liste befindet sich ein (binärer) balancierter Suchbaum. Die Listenelemente sind zugleich die Blätter dieses Baumes. Der Suchbaum erlaubt nicht nur eine normale, bei der Wurzel beginnende Suche von oben nach unten. Man kann mit einer Suche auch an den Blättern bei der Position beginnen, auf die der Finger zeigt. Von dort läuft man das rechte Rückgrat des Baumes hinauf solange, bis man erstmals bei einem Knoten angekommen ist, der die Wurzel eines Teilbaumes mit der gesuchten Stelle ist. Von diesem Knoten aus wird wie üblich abwärts gesucht, bis man die gesuchte Stelle (unter den Blättern) gefunden hat. Fügt man jetzt ein neues Element in die Liste ein, muss man unter Umständen die darüber befindliche Suchstruktur rebalancieren. Das kann zu einem erneuten Hinaufwandern von der Einfügestelle bis (schlimmstenfalls) zur Wurzel führen.

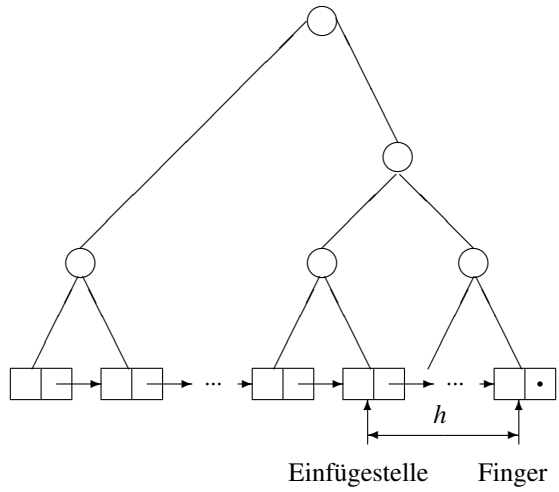



**Abbildung 2.11**

Falls man die richtigen „Wegweiser“ an den inneren Knoten des Suchbaumes postiert, kann somit die Suche nach einer $h$ Elemente von der Position des Fingers am Ende entfernten Einfügestelle stets in $O(\log(h+1))$ Schritten ausgeführt werden. Die Suche geht damit immer schnell. Zwar kann das Einfügen an der richtigen Stelle in der Liste in $O(1)$ Schritten ausgeführt werden; das anschließende Rebalancieren des Suchbaums kann aber $\Omega(\log N)$ Schritte im ungünstigsten Fall kosten.

Glücklicherweise tritt dieser ungünstige Fall für die meisten Typen balancierter Bäume nicht allzu häufig ein. Genauer gilt etwa für AVL-Bäume und 1-2-Bruder-Bäume: Die über eine Folge von $N$ iterierten Einfügungen in den anfangs leeren Baum amortisierten Rebalancierungskosten (ohne Suchkosten) sind im schlechtesten Fall $O(N)$. D. h. als Folge einer einzelnen Einfügeoperation kann zwar Zeit $\Omega(\log N)$ erforderlich

sein um den Baum zu rebalancieren, der mittlere Rebalancierungsaufwand pro Einfügeoperation, gemittelt über eine beliebige Folge von Einfügeoperationen in den anfangs leeren Baum, ist aber konstant.

Das Sortierverfahren verläuft daher so: Die $N$ Elemente der gegebenen Folge $F = \langle k_1, \ldots, k_N \rangle$ werden der Reihe nach in die anfangs leere Struktur der oben beschriebenen Art eingefügt. Wir nennen das Verfahren *A-sort* für **a**daptives Sortieren oder AVL-Sortieren (vgl. [135]).

Bezeichnet wieder $h_j$ den Abstand des Folgenelementes $k_j$ vom jeweiligen Listenende, also von der Fingerposition, so gilt für die Gesamtlaufzeit von A-sort offensichtlich:

$$T(F) = \underbrace{O(N)}_{\text{Umstrukturierungsaufwand}} + \underbrace{O\left(\sum_{j=1}^{N} \log(h_j+1)\right)}_{\text{gesamter Suchaufwand}}$$

Um die Zeit $T(F)$ zum Sortieren einer Folge $F$ mit der Anzahl der Inversionen von $F$ in Verbindung bringen zu können, beachten wir, dass $inv(F) = \sum_{j=1}^{N} h_j$ gilt und:

$$\begin{aligned}
\sum_{j=1}^{N} \log(h_j+1) &= \log\left(\prod_{j=1}^{N}(h_j+1)\right) \\
&= N\log\left(\prod_{j=1}^{N}(h_j+1)^{\frac{1}{N}}\right) \\
&\overset{\{*\}}{\leq} N\log\left(\sum_{j=1}^{N}\frac{(h_j+1)}{N}\right) \\
&= N\log\left(1+\frac{\sum_{j=1}^{N} h_j}{N}\right) \\
&= N\log\left(1+\frac{inv(F)}{N}\right)
\end{aligned}$$

In $\{*\}$ wurde die Tatsache benutzt, dass das arithmetische Mittel nie kleiner sein kann als das geometrische Mittel der Größen $(h_j+1),\ j = 1,\ldots,N$.

Damit folgt insgesamt, dass das Verfahren A-sort jede Folge $F$ der Länge $N$ in Zeit

$$T(F) = O\left(N + N\log\left(1+\frac{inv(F)}{N}\right)\right)$$

sortiert. Die Laufzeit ist also linear, solange $inv(F) \in O(N)$ bleibt und für die maximale Inversionszahl $N(N-1)/2$ wie zu erwarten $O(N\log N)$.

Ist das Verfahren *inv*-optimal? Nach der im Abschnitt 2.6.1 gegebenen Definition genügt es dazu zu zeigen, dass

$$\log\left(\left|\left\{F' \mid inv(F') \leq inv(F)\right\}\right|\right) \in \Omega\left(N \cdot \log\left(1+\frac{inv(F)}{N}\right)\right)$$

ist. Man muss also zeigen, dass der Logarithmus der Anzahl der Permutationen einer Folge von $N$ Elementen mit höchstens $I(N)$ Inversionen von der Größenordnung $\Omega(N\log(1+\frac{I(N)}{N}))$ ist. Dies ist tatsächlich der Fall; wir verweisen dazu auf [128].

Wie verhält sich A-sort, wenn man ein anderes Maß für die Vorsortierung wählt? Wir zeigen, dass A-sort nicht *runs*-optimal ist, indem wir nachweisen:

(a) Falls A-sort *runs*-optimal ist, muss A-sort alle Folgen mit nur zwei Runs in linearer Zeit sortieren.

(b) Es gibt eine Folge mit nur zwei Runs, für die das Verfahren A-sort $\Omega(N\log N)$ Zeit benötigt.

Zum Beweis von (a) schätzen wir die Anzahl der Permutationen von $N$ Zahlen mit höchstens zwei Runs ab. Offenbar kann man jede solche Permutation mit höchstens zwei Runs erzeugen, indem man eine der $2^N$ möglichen Teilmengen der $N$ Zahlen nimmt, sie aufsteigend sortiert und daraus einen Run bildet. Der zweite Run besteht aus der aufsteigend sortierten Folge der übrig gebliebenen Elemente. Also folgt, dass es höchstens $O(2^N)$ Permutationen von $N$ Elementen mit nur zwei Runs geben kann. Nach der Definition in 2.6.1 muss ein *runs*-optimaler Algorithmus $A$ zumindest jede Folge $F$ mit höchstens zwei Runs sortieren in Zeit

$$\begin{aligned} T_A(F,runs) &\leq c\cdot\left(N+\log\left(\left|\left\{F' \mid runs(F')\leq 2\right\}\right|\right)\right) \\ &= c\cdot(N+\log 2^N) = O(N). \end{aligned}$$

Zum Nachweis von (b) betrachten wir die Folge

$$F = \Big\langle \underbrace{\frac{N}{2}+1, \frac{N}{2}+2, \ldots, N}_{\frac{N}{2}}, \underbrace{1,2,\ldots,\frac{N}{2}}_{\frac{N}{2}} \Big\rangle$$

(Hier nehmen wir ohne Einschränkung an, dass $N$ gerade ist.) Es ist $runs(F)=2$, aber $inv(F)=\Theta(N^2)$. Genauer gilt für die Größen $h_j$, die die Laufzeit des Verfahrens A-sort bestimmen, in diesem Fall:

$$h_j = 0, \text{ für } 1\leq j\leq \frac{N}{2}, \text{ und } h_j = \frac{N}{2}, \text{ für } \frac{N}{2} < j \leq N$$

Die Laufzeit von A-sort für diese Folge ist wenigstens gleich dem gesamten Suchaufwand und Umstrukturierungsaufwand, also:

$$\begin{aligned} \sum_{j=1}^{N} c\,(1+\log(h_j+1)) &= cN + c\cdot\log\prod_{j=\frac{N}{2}+1}^{N}\left(1+\frac{N}{2}\right) \\ &= cN + c\cdot\log\left(1+\frac{N}{2}\right)^{\frac{N}{2}} \\ &= \Omega(N\log N) \end{aligned}$$

□

Es ist intuitiv klar, warum A-sort für diese Folge $F$ so viel Zeit benötigt. Alle Zahlen $1,2,\ldots,\frac{N}{2}$ müssen relativ weit von der festen Position des Fingers am rechten Ende der jeweiligen Liste eingefügt werden. Es wäre daher sicher besser, wenn man die Suche nach der jeweils nächsten Einfügestelle nicht immer wieder dort starten würde, sondern den Finger mitbewegen würde.

### 2.6.3 Sortieren durch lokales Einfügen und natürliches Verschmelzen

Wir haben bereits am Ende des vorigen Abschnitts gesehen, dass die Implementation des Sortierens durch Einfügen mithilfe dynamischer, sortierter Listen mit einem fest gehaltenen Finger am rechten Ende nicht immer zu einem $m$-optimalen Verfahren führt. Es ist nahe liegend die Einfügestrategie wie folgt zu verändern: Man fügt die Elemente der gegebenen, zu sortierenden Folge der Reihe nach in die anfangs leere und stets aufsteigend sortierte Liste ein; zur Bestimmung der Einfügestelle für das jeweils nächste Element startet man eine Suche aber nicht jedes Mal von derselben, festen Position am Listenende, sondern von der Position, an der das letzte Element eingefügt wurde. Man benötigt also eine Struktur mit einem *beweglichen Finger*. Der Finger zeigt stets auf die Position des jeweils zuletzt eingefügten Elementes. Das ist der Ausgangspunkt für die Suche nach der richtigen Einfügestelle für das jeweils nächste einzufügende Element.

Der Aufwand für die Suche nach der jeweils nächsten Einfügestelle hängt also entscheidend ab von der Distanz zweier nacheinander einzufügender Elemente der Ausgangsfolge. Für eine Folge $F = \langle k_1,\ldots,k_n\rangle$ ist die Distanz $d_j$ des Elementes $k_j$ vom vorangehenden offenbar:

$$d_j = \left|\left\{i \,|\, 1 \le i < j \text{ und } (k_{j-1} < k_i < k_j \text{ oder } k_j < k_i < k_{j-1})\right\}\right|$$

**Beispiel:** Für die Folge $F = \langle 5,1,7,4,9,2,8,3,6\rangle$ ($F_c$ aus 2.6.1) hat das siebente Element $k_7 = 8$ die Distanz $d_7 = 3$ vom vorangehenden Element $k_6 = 2$, da links von $k_7$ drei Elemente der Folge $F$ stehen, deren Platz bei aufsteigender Sortierung zwischen $k_6$ und $k_7$ ist.

Für gut vorsortierte Folgen wird man erwarten, dass die Distanzen $d_j$ klein sind. Ein Vorsortierung berücksichtigendes Sortierverfahren kann sich das zu Nutze machen, wenn es der allgemeinen Strategie des Sortierens durch Einfügen folgt und dies Verfahren durch eine Struktur mit einem beweglichen Finger implementiert ist, der auf das jeweils zuletzt eingefügte Element zeigt.

Man möchte natürlich erreichen, dass die Suche nach der nächsten Einfügestelle für ein Element mit Distanz $d$ von der Finger-Position möglichst in $O(\log d)$ Schritten ausführbar ist. Das kann eine zu der im Abschnitt 2.6.2 beschriebenen analoge Hybridstruktur leisten, die aus einer dynamischen, verketteten sortierten Liste mit darübergestülptem Suchbaum besteht, wenn der Suchpfad immer wie in Abbildung 2.12 aussieht.

Leider kann es aber vorkommen, dass zwei ganz nah benachbarte Blätter eines (binären) Suchbaumes nur durch einen Pfad mit Länge $\Omega(\log N)$, wobei $N$ die gesamte Anzahl aller Blätter ist, miteinander verbunden sind, wie Abbildung 2.13 zeigt. Man

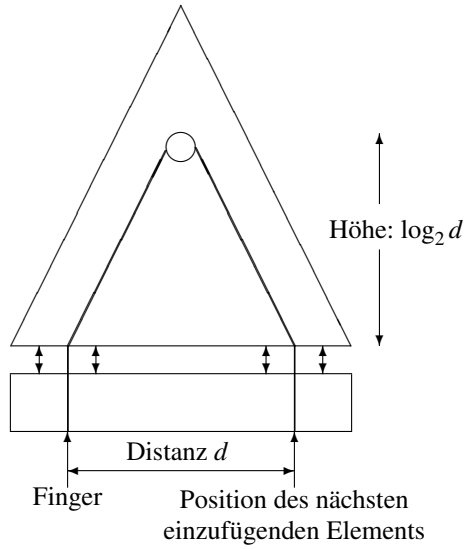



**Abbildung 2.12**

wird also, anders als im Fall eines festen Fingers am rechten Ende, nicht immer erwarten können, dass man zur Bestimmung einer Einfügestelle mit Distanz $d$ nur $O(\log d)$ Niveaus im Suchbaum hinauf- und auch wieder hinabsteigen muss, wenn man nicht zusätzliche Verbindungen der Knoten untereinander vorsieht. Mit einer niveauweisen Verkettung benachbarter Knoten im Baum kann man das Problem allerdings lösen. Es gibt in der Literatur zahlreiche Vorschläge für solche Strukturen.

Wir verweisen auf das Kapitel über Bäume und auf [135] und begnügen uns mit der (hoffentlich plausiblen) Feststellung, dass es eine Struktur mit einem beweglichen Finger gibt, die zum *Sortieren durch lokales Einfügen* wie folgt verwendet werden kann: Die zu sortierenden Elemente werden der Reihe nach in die anfangs leere Struktur eingefügt. Die bereits eingefügten Elemente sind aufsteigend sortiert und miteinander verkettet. Ein Finger zeigt auf das jeweils zuletzt eingefügte Element. Die Stelle, an der das jeweils nächste Element mit Distanz $d$ in die verkettete, aufsteigend sortierte Liste der bereits betrachteten Elemente eingefügt werden muss, kann in Zeit $O(\log d)$ gefunden werden. Die zur Rebalancierung der Suchstruktur nach einer Einfügung erforderliche Schrittzahl ist im Durchschnitt (genommen über eine Folge von Einfügungen in die anfangs leere Struktur) konstant. Damit folgt, dass das Verfahren Sortieren durch lokales Einfügen eine Folge $F = \langle k_1, \ldots, k_N \rangle$ mit Distanzen $d_j,\ 1 \leq j \leq N$, stets in Zeit

$$T(F) = O(N) + O\left(\sum_{j=1}^{N} \log(1+d_j)\right)$$

Schritten zu sortieren erlaubt.

Es ist intuitiv sofort plausibel, dass Sortieren durch lokales Einfügen für Folgen mit kleiner Inversionszahl eher besser sein muss als das im Abschnitt 2.6.2 vorgestellte Ver-

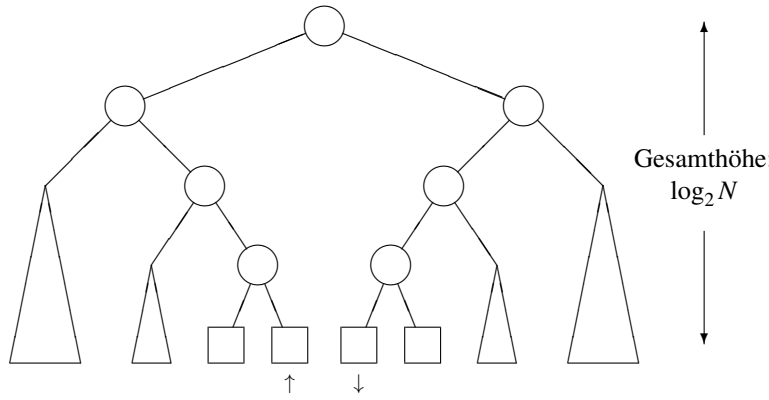



**Abbildung 2.13**

fahren A-sort. Denn für jede Folge $F = \langle k_1, \ldots, k_N \rangle$ gilt: Die Distanz $d_j$ eines Elementes vom vorangehenden kann nicht größer sein als das Maximum der Abstände von $k_j$ und $k_{j-1}$ vom rechten Ende. Genauer: Die Anzahl der links von $k_j$ in $F$ stehenden Elemente, die in der sortierten Folge zwischen $k_{j-1}$ und $k_j$ gehören, ist kleiner als die größere der Anzahlen $h_j$ und $h_{j-1}$ von Elementen, die links von $k_j$ bzw. links von $k_{j-1}$ in $F$ stehen und größer als $k_j$ bzw. $k_{j-1}$ sind. Daher folgt:

$$d_j + 1 \leq \max(h_{j-1}, h_j) + 1 \leq (h_j + 1)(h_{j-1} + 1)$$

und

$$\begin{aligned} \sum_{j=1}^{N} \log(1 + d_j) &\leq \sum_{j=1}^{N} \log\left((h_j + 1)(h_{j-1} + 1)\right) \\ &\leq 2 \sum_{j=1}^{N} \log(h_j + 1). \end{aligned}$$

Es ergibt sich also wie im Falle von A-sort

$$T(F) = O\left(N + N \log\left(1 + \frac{inv(F)}{N}\right)\right)$$

als Abschätzung für die Laufzeit des Sortierens durch lokales Einfügen. Das Verfahren Sortieren durch lokales Einfügen ist damit ebenfalls *inv*-optimal.

H. Mannila [128] hat gezeigt, dass Sortieren durch lokales Einfügen auch *runs*-optimal und *rem*-optimal ist. In beiden Fällen verläuft der Beweis analog zur Argumentation im Falle der *inv*-Optimalität in zwei Schritten. Zuerst wird die Laufzeit des Verfahrens für alle Folgen mit gegebenem Vorsortierungsgrad, gemessen mit den Maßen *runs* und *rem*, abgeschätzt (Worst-case-Zeitschranke). Dann werden untere Schranken für die Mindestanzahl von Permutationen von $N$ Elementen und gegebenem Vorsortierungsgrad angegeben. Genauer wird gezeigt:

**Satz 2.2** *Jede Folge $F$ von $N$ Schlüsseln kann nach dem Verfahren Sortieren durch lokales Einfügen sortiert werden in*

$(a)$ $O(N(1+\log(runs(F))))$

*und*

$$(b)\ O\left(N+\sum_{j=0}^{rem(F)-1}\log(N-j)\right)$$

*Schritten.*

**Satz 2.3**

$(a)$ *Es gibt Konstanten $c$ und $d$ derart, dass gilt:*

$$\log(|\{F \mid runs(F) \leq t\}|) \geq c \cdot N \cdot \log t - d \cdot t$$

$$(b)\ \log(|\{F \mid rem(F) \leq s\}|) \geq \sum_{j=0}^{s-1}\log(N-j)$$

Für die nicht einfachen Beweise von Satz 2.2 (a), (b) und Satz 2.3 (a) verweisen wir auf die Arbeit von H. Mannila [128]. Wir begnügen uns damit, die einfach nachzuweisende untere Schranke in Satz 2.3 (b) herzuleiten.

Wie viele Permutationen $F$ von $N$ Schlüsseln mit $rem(F) \leq s$, also mit einer wenigstens $N-s$ langen aufsteigenden Teilfolge, gibt es mindestens? Wir können Folgen mit $rem(F) \leq s$ bilden, indem wir zunächst $N-s$ Elemente aus den $N$ Elementen auswählen, diese Elemente in aufsteigende Reihenfolge bringen und die verbleibenden $s$ Elemente in beliebiger Reihenfolge dahinter schreiben. Also gibt es wenigstens

$$\binom{N}{N-s}s! = \frac{N!}{(N-s)!} = \prod_{j=0}^{s-1}(N-j)$$

derartige Folgen. Also ist

$$\log(|\{F \mid rem(F) \leq s\}|) \geq \log\prod_{j=0}^{s-1}(N-j) = \sum_{j=0}^{s-1}\log(N-j)$$

und Satz 2.3 (b) ist bewiesen. Aus den beiden Sätzen folgt sofort, dass Sortieren durch lokales Einfügen *runs*-optimal und *rem*-optimal im Sinne der Definition aus Abschnitt 2.6.1 ist.

Dass das geringe Distanzen zwischen aufeinander folgenden Folgenelementen ausnutzende Verfahren Sortieren durch lokales Einfügen *runs*-optimal ist, ist durchaus überraschend (und nicht leicht zu zeigen). Es gibt ein Verfahren, für das man wesentlich eher vermuten würde, dass es *runs*-optimal ist. Das ist das *Sortieren durch natürliches Verschmelzen* (vgl. Abschnitt 2.4.3). Es sortiert eine Folge $F$ nach folgender Methode: Man verschmilzt je zwei der ursprünglich in $F$ vorhandenen „natürlichen“ Runs zu je einem neuen Run. Das ergibt eine Folge mit nur noch etwa halb so vielen Runs wie in

der ursprünglich gegebenen Folge. Auf die neue Folge wendet man dasselbe Verfahren an usw., bis man schließlich eine nur noch aus einem einzigen Run bestehende und damit sortierte Folge erhalten hat.

Betrachten wir als Beispiel die Folge $F = F_c$ aus Abschnitt 2.6.1:

$$5 \mid 1\ 7 \mid 4\ 9 \mid 2\ 8 \mid 3\ 6 \mid$$

Die Folge hat fünf „natürliche“ Runs, die wir durch vertikale Striche voneinander getrennt haben. Verschmelzen des ersten und zweiten sowie des dritten und vierten Runs ergibt:

$$1\ 5\ 7 \mid 2\ 4\ 8\ 9 \mid 3\ 6$$

Verschmelzen der ersten beiden Runs ergibt:

$$1\ 2\ 4\ 5\ 7\ 8\ 9 \mid 3\ 6$$

Abermaliges Verschmelzen der verbliebenen zwei Runs liefert die aufsteigend sortierte Folge.

Bezeichnet man das Herstellen einer neuen Folge durch paarweises Verschmelzen je zweier benachbarter Runs als einen *Durchgang*, so ist klar, dass jeder Durchgang in linearer Zeit ausführbar ist und höchstens $O(\log t)$ Durchgänge zum Sortieren ausgeführt werden müssen, wenn $t$ die Anzahl der ursprünglich vorhandenen Runs ist. Damit erhält man als Laufzeit des Verfahrens für beliebige Folgen der Länge $N$ unmittelbar

$$T(F) = O\left(N\left(1 + \log\left(\mathit{runs}(F)\right)\right)\right).$$

Zusammen mit Satz 2.3 (a) folgt, dass Sortieren durch natürliches Verschmelzen *runs*-optimal ist.

Abschließend noch eine Bemerkung zum Speicherbedarf aller in diesem Abschnitt besprochenen Sortierverfahren. A-sort, Sortieren durch lokales Einfügen und natürliches Verschmelzen sortieren nicht „am Ort“. Sie benötigen $\Theta(N)$ zusätzlichen Speicherplatz zum Sortieren von Folgen der Länge $N$. Im Falle der grob skizzierten Hybridstrukturen kann der in $\Theta(N)$ versteckte Faktor beträchtlich sein.

## 2.7 Externes Sortieren

In den bisherigen Abschnitten über das Sortieren sind wir stets davon ausgegangen, dass die zu sortierenden Daten und alle Zwischenergebnisse im Hauptspeicher des verwendeten Rechners Platz finden. Programmtechnisch haben wir diese Situation durch die Datenstruktur des Arrays modelliert. Der Zugriff auf einen Datensatz hat stets konstante Zeit beansprucht, unabhängig davon, um welchen Datensatz es ging. Entsprechend haben wir unsere Sortieralgorithmen nicht darauf abgestellt, die Datensätze in einer gewissen Reihenfolge zu betrachten.

Die Situation ist beim Sortieren von Datensätzen auf Externspeichern (Sekundärspeichern) grundlegend anders. Gängige Externspeicher heutiger Rechner sind vor allem

Magnetplatten, CDROM und Magnetbänder. Wir wollen in diesem Abschnitt lediglich das Sortieren mit Magnetbändern betrachten, weil hier die Restriktionen am stärksten sind. Die vorgestellten Verfahren können auch für Platten oder wiederbeschreibbare CDs benutzt werden, obwohl sie die dort verfügbaren Operationen nicht voll ausschöpfen. Im Wesentlichen kann man die Datensätze auf ein Band sequenziell schreiben oder von dort sequenziell lesen. Der Zugriff auf einen Datensatz nahe am Bandende ist, wenn man gerade auf einen Datensatz am Bandanfang zugegriffen hat, sehr aufwändig. Es muss auf sämtliche Datensätze zwischen dem aktuellen und dem gewünschten ebenfalls zugegriffen werden. Dieses Modell präzisieren wir im Abschnitt 2.7.1.

Man wird versuchen Sortieralgorithmen an diese Situation anzupassen indem man Datensätze in derjenigen Reihenfolge bearbeitet, in der sie schon auf dem Magnetband stehen. Man kennt heute viele verschiedene Verfahren, die dieser Idee folgen. Weil in der Anfangszeit der elektronischen Rechner Internspeicher noch knapper und teurer war als heute, hat man sich bereits sehr früh intensiv mit Methoden des externen Sortierens beschäftigt. D. Knuth [100] widmet diesem Thema weit über hundert Seiten und berichtet über erste externe Sortierverfahren aus dem Jahr 1945 (von J. P. Eckert und J. W. Mauchly), unter anderem über das *ausgeglichene 2-Wege-Mergesort* für Magnetbänder, das wir in Abschnitt 2.7.2 vorstellen. Das *ausgeglichene Mehr-Wege-Mergesort*, das es erlaubt mehrere Bänder in den Sortierprozess einzubeziehen wird im Abschnitt 2.7.3 erläutert. Schließlich präsentieren wir im Abschnitt 2.7.4 das *Mehrphasen-Mergesort*. All diese Verfahren sind Varianten des Sortierens durch Verschmelzen. Grundsätzlich eignen sich auch andere Methoden, wie etwa Radixsort oder Quicksort, zum externen Sortieren (vgl. [100] und [185]); wir wollen uns hier aber auf die gebräuchlicheren Mergesort-Varianten beschränken.

## 2.7.1 Das Magnetband als Externspeichermedium

Das Speichermedium Magnetband ähnelt in vieler Hinsicht den Magnetbändern in Audio-Kassetten; an viele Mikrorechner kann man ja heute einfache Kassettenrecorder als Externspeichergeräte anschließen. Datensätze werden auf einem Magnetband streng sequenziell gespeichert. Mit einem Magnetband können folgende Operationen ausgeführt werden:

**Zurückspulen und Lese- oder Schreibzustand wählen:** Das Band wird an den Anfang zurückgespult. Beim sequenziellen Bearbeiten des Bandes können entweder nur Einträge vom Band gelesen oder nur Einträge auf das Band geschrieben werden. Daher wird, zusammen mit dem Zurückspulen, entweder der Lese- oder der Schreibzustand gewählt. (Nach einer weiteren Rückspuloperation kann dann für dasselbe Band ein anderer Zustand gewählt werden.) Wir wählen folgende Bezeichnungen für diese Operationen:

| | |
|---|---|
| $reset(t)$ : | Rückspulen des Bandes $t$ (englisch: tape) mit Wahl des Lesezustands; |
| $rewrite(t)$ : | Rückspulen des Bandes $t$ mit Wahl des Schreibzustandes. |

**Lesen oder Schreiben:** Das Lesen ist das Übertragen des nächsten Datensatzes vom Band in den Internspeicher des Rechners. Das Band muss sich im Lesezustand befinden. Entsprechend ist das Schreiben das Übertragen eines Datensatzes vom Internspeicher an die nächste Stelle auf dem Band, wobei das Band im Schreibzustand sein muss. Der Internspeicherbereich, in den gelesen bzw. von dem geschrieben wird, wird ebenfalls spezifiziert, und zwar einfach durch Angabe einer Variablen für den Datensatz:

| | |
|---|---|
| $read(t,d)$ : | Lesen des nächsten Datensatzes vom Band $t$ und Zuweisen an die Variable $d$; |
| $write(t,d)$ : | Schreiben des Werts der Variablen $d$ als nächsten Datensatz auf Band $t$. |

Magnetbänder haben in der Realität nur eine endliche Länge; wir wollen hier von dieser Restriktion abstrahieren und Bänder als einseitig unendliche Datenspeicher ansehen. Wir müssen also nicht befürchten beim Lesen oder Beschreiben eines Bandes das Bandende zu erreichen. Beim Lesen werden wir aber sehr wohl das Ende der Datensätze erreichen, die dort auf dem Band gespeichert sind. Eine Funktion liefert uns die nötige Information.

**Feststellen, ob das Ende der Datensätze erreicht ist:** Diese Funktion verwenden wir nur im Lesezustand. Sobald das Ende erreicht ist, sind alle Datensätze gelesen worden. Das weitere Lesen eines Datensatzes ist dann nicht mehr sinnvoll. Wir bezeichnen diese Funktion mit *eof* (englisch: *end of file*):

| | |
|---|---|
| $eof(t)$ : | liefert den Wert *true* genau dann, wenn das Datenende für Band $t$ erreicht ist, und sonst den Wert *false*. |

Wir haben die Namen für Operationen und Funktionen bewusst gewählt wie die Datei-Bearbeitungs-Prozeduren und -Funktionen in Pascal, weil das Datei-Konzept in Pascal gerade Magnetbänder modelliert. Moderne Magnetbänder erlauben über die genannten Operationen hinaus etwa das Rückwärts-Lesen, das Rückwärts-Schreiben, Lesen mit Schreiben kombiniert und andere mehr, die natürlich von passenden Sortierverfahren durchaus genutzt werden können. Wir lassen dieses aber hier unberücksichtigt, damit der Grundgedanke externen Sortierens möglichst deutlich zu Tage tritt.

Typischerweise sind Ein-/Ausgabe-Operationen (Externzugriffe) mit Sekundärspeichern erheblich langsamer als interne Operationen, sei es das Umspeichern von Daten im Hauptspeicher oder das Anstellen von Berechnungen. Schnelle externe Sortierverfahren müssen daher in erster Linie bemüht sein die Anzahl der Externzugriffe so gering wie möglich zu halten. Das Komplexitätsmaß, das wir für externe Sortierverfahren mit Bändern verwenden, basiert daher auf der Anzahl der Externzugriffe. Weil bei Bändern Externzugriffe nur rein sequenziell möglich sind, zählen wir, wie oft ein ganzes Band gelesen oder geschrieben wurde. Die Kosten für das Lesen oder Schreiben eines Bandes sind proportional zur Anzahl der Datensätze auf diesem Band. Sind alle Datensätze auf mehrere Bänder verteilt, so ergeben sich beim Lesen all dieser Bänder Gesamtkosten, die proportional zur Anzahl der Datensätze insgesamt sind. Das Lesen/Schreiben aller Datensätze nennen wir einen *Durchgang* (englisch: *pass*). Als Komplexitätsmaß wählen wir die Anzahl der Durchgänge, die zum Sortieren von $N$ Datensätzen benötigt werden; wir bezeichnen die minimale, mittlere und maximale Anzahl mit $P_{min}(N)$, $P_{mit}(N)$ und $P_{max}(N)$.

### 2.7.2 Ausgeglichenes 2-Wege-Mergesort

Die zu sortierenden $N$ Datensätze stehen auf einem Magnetband, dem Eingabe-Band $t_1$. Die Anzahl $N$ der Datensätze ist so groß, dass nicht alle Datensätze gleichzeitig im Hauptspeicher Platz finden. Den externen Mergesort-Varianten liegt nun folgende nahe liegende Idee zu Grunde. Man teilt die $N$ Datensätze gedanklich in $\lceil \frac{N}{I} \rceil$ Teilfolgen von jeweils höchstens $I$ Datensätzen, wobei $I$ die Anzahl der Datensätze ist, die höchstens im Internspeicher Platz finden. Dann betrachtet man der Reihe nach jede dieser Teilfolgen separat. Man liest die Teilfolge ganz in den Internspeicher ein, sortiert sie mit einem der bereits bekannten Verfahren und schreibt die sortierte Teilfolge auf den Externspeicher.

Schließlich verschmilzt man die sortierten Teilfolgen. Das Verschmelzen kann effizient mithilfe des Externspeichers geschehen, weil sowohl die zu verschmelzenden Teilfolgen als auch die entstehende Resultatfolge rein sequenziell gelesen bzw. geschrieben werden. Die zu verschmelzenden Teilfolgen müssen dazu auf verschiedenen Bändern stehen. Anfangs muss man also die Datensätze auf dem Eingabeband auf mehrere Bänder aufteilen; danach verschmilzt man die sortierten Teilfolgen. Die entstandene Folge muss wieder aufgeteilt werden usw., bis man schließlich eine vollständig sortierte Folge erzeugt hat. Dieser Wechsel zwischen *Aufteilungs-* und *Verschmelzungsphase* ist charakteristisch für externe Mergesort-Verfahren. Wohl das einfachste solche Verfahren ist das *ausgeglichene 2-Wege-Mergesort* (*balanced 2-way-mergesort*), das wir jetzt vorstellen.

**Methode:** Wir verwenden vier Bänder, $t_1, t_2, t_3, t_4$. Das Eingabeband ist $t_1$. Es werden wiederholt $I$ Datensätze von $t_1$ gelesen, intern sortiert, und abwechselnd solange auf $t_3$ und $t_4$ geschrieben, bis $t_1$ erschöpft ist. Dann stehen also $\lceil N/(2 \cdot I) \rceil$ sortierte Teilfolgen (Runs) der Länge $I$ auf $t_3$ und $\lfloor N/(2 \cdot I) \rfloor$ Runs der Länge $I$ auf $t_4$. Jetzt werden die Runs von $t_3$ und $t_4$ verschmolzen. Dabei entstehen Runs der Länge $2 \cdot I$. Diese werden abwechselnd auf $t_1$ und $t_2$ verteilt. Dann stehen also etwa $N/(4 \cdot I)$ Runs der Länge $2 \cdot I$ auf jedem der Bänder $t_1, t_2$. Nach jeder Aufteilungs- und Verschmelzungsphase hat sich die Run-Länge verdoppelt und die Anzahl der Runs etwa halbiert. Wir fahren fort die Runs von zweien der Bänder zu verschmelzen und abwechselnd auf die anderen beiden zu verteilen, dann die Bänder (logisch) zu vertauschen bis schließlich nur noch ein Run auf einem der Bänder übrig bleibt.

Das Verschmelzen zweier externer Runs kann mithilfe zweier Variablen für die beiden Bänder geschehen, also mit ganz wenig internem Speicherplatz. Beim internen Sortieren zu Beginn wird der gesamte interne Speicherplatz genutzt um möglichst lange Runs zu erzielen. Betrachten wir ein Beispiel.

**Beispiel:** Der Hauptspeicher des Rechners fasse drei Datensätze ($I = 3$) und die Folge der zu sortierenden Schlüssel sei

$$F = 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.$$

Anfangs stehen die Datensätze, hier repräsentiert durch ihre Schlüssel, auf Band $t_1$; alle anderen Bänder sind leer:

$t_1$ : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.
$t_2$ :
$t_3$ :
$t_4$ :

Zunächst werden jeweils $I$ Datensätze von $t_1$ gelesen, sortiert, und abwechselnd auf $t_3$ und $t_4$ aufgeteilt. Nach den ersten drei gelesenen Datensätzen ergibt sich folgendes Bild:

$t_1$ : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.
$t_2$ : __
$t_3$ : 2, 5, 12 __
$t_4$ : __

Die Stelle des nächsten Datensatzes jedes Bandes ist unterstrichen. Nachdem alle Datensätze auf $t_3$ und $t_4$ verteilt sind, haben wir folgende Situation (Runs sind durch ; getrennt):

$t_1$ : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8. __
$t_2$ : __
$t_3$ : 2, 5, 12; 1, 4, 14; 7, 8, 11. __
$t_4$ : 6, 13, 15; 3, 9, 10. __

Alle Bänder werden zurückgespult, $t_3$ und $t_4$ werden gelesen, $t_1$ und $t_2$ beschrieben. Der ursprüngliche Inhalt von $t_1$ ist damit verloren. Wir zeigen die Situationen, die sich jeweils nach einer Verschmelzungs- und Verteilungsphase ergeben:

$t_1$ : 2, 5, 6, 12, 13, 15; 7, 8, 11. __
$t_2$ : 1, 3, 4, 9, 10, 14. __
$t_3$ : __
$t_4$ : __

$t_1$ : __
$t_2$ : __
$t_3$ : 1, 2, 3, 4, 5, 6, 9, 10, 12, 13, 14, 15. __
$t_4$ : 7, 8, 11. __

$t_1$ : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. __
$t_2$ : __
$t_3$ : __
$t_4$ : __

Dass die sortierte Folge auf Band $t_1$ entstanden ist, ist hierbei Zufall. Im Allgemeinen kann sie auf Band $t_1$ oder $t_3$ entstehen, wenn von den beiden Bändern $t_1$ und $t_2$ (bzw. $t_3$ und $t_4$) entstehende Runs zuerst auf $t_1$ und danach auf $t_2$ (bzw. zuerst auf $t_3$ und danach auf $t_4$) geschrieben werden.

Zur programmtechnischen Realisierung dieses Verfahrens setzen wir die folgenden Definitionen voraus:

```
const
  k = 4; {Anzahl der Bänder}
type
  tape = file of item;
  tapes = array [1 .. k] of tape
```

Dann lässt sich die Sortierprozedur wie folgt angeben:

```
procedure balanced_2_way_mergesort (var t : tapes; var i : integer);
  {sortiert die Datensätze von Band t[1] auf eines der Bänder t[i],
   und liefert Band-Nummer i zurück}
  var
    ein1, ein2, aus1, aus2, aus: integer;
  begin
    anfangsverteilung({von :} t[1], {nach:} t[3], t[4]);
    {Runs der Länge 1 sind auf t[3] und t[4] verteilt}
    {wähle 2 Ein- und 2 Ausgabebänder:}
    ein1 := 3;
    ein2 := 4;
    aus1 := 1;
    aus2 := 2;
    {falls ein2 leer ist, steht die sortierte Folge auf ein1}
    reset(t[ein2]);
    while not eof(t[ein2]) do
      begin
        aus := aus2; {zuletzt benutztes Ausgabeband}
        reset(t[ein1]);
        rewrite(t[aus1]);
        rewrite(t[aus2]);
        while not eof(t[ein2]) do
          begin {ein2 wird zuerst erschöpft}
            nächstes(aus); {welches Ausgabeband}
            mergeruns(t[ein1], t[ein2], t[aus])
            {verschmilz je einen Run aus Band ein1
             und Band ein2 und schreibe ihn auf Band aus}
          end;
        copyrest(t[ein1], t[aus]); {falls noch ein Run auf ein1}
        {ein1 und ein2 sind erschöpft; wechsle Ein-/Ausgabebänder}
        tausche(aus1, ein1);
        tausche(aus2, ein2);
        reset(t[ein2])
      end;
    {jetzt steht die sortierte Folge auf t[ein1]}
    i := ein1
  end
```

Wir geben noch kurz an, was die bisher nicht erklärten Prozeduren leisten ohne sie aber im Detail auszufüllen; das überlassen wir dem interessierten Leser.

**procedure** *anfangsverteilung* (**var** $t_1, t_3, t_4$ : *tape*);
{*sortiert Teilfolgen der Länge I, die aus* $t_1$ *stammen, und speichert sie abwechselnd nach* $t_3$ *und* $t_4$*, beginnend mit* $t_3$*; dort werden also Runs der Länge I gespeichert*}

**procedure** *nächstes* (**var** *aus* : *integer*);
{*wechselt das Ausgabeband, also* $1 \to 2,\ 2 \to 1,\ 3 \to 4,\ 4 \to 3$}

**procedure** *mergeruns* (**var** $t_1, t_2, t$ : *tape*);
{*verschmilzt je einen Run aus* $t_1$ *und* $t_2$ *und schreibt den doppelt langen verschmolzenen Run auf* $t$; *beim Verschmelzen werden gleichzeitig lediglich zwei Datensätze intern gespeichert, je einer aus* $t_1$ *und* $t_2$}

**procedure** *copyrest* (**var** $t_1, t$ : *tape*);
{*überträgt den Rest der Datensätze von* $t_1$ *nach* $t$}

**procedure** *tausche* (**var** $i, j$ : *integer*);
{*vertauscht die Werte von i und j*}

**Analyse:** Nach jeder Verschmelzungs- und Verteilungsphase hat sich die Anzahl der Runs (etwa) halbiert. In der Anfangsverteilung haben wir aus $N$ Datensätzen unter Zuhilfenahme des Internspeichers der Größe $I$ in einem Durchgang $\lceil \frac{N}{I} \rceil$ Runs hergestellt. Damit ergibt sich nach $\lceil \log(\frac{N}{I}) \rceil$ Durchgängen ein einziger Run; also ist

$$P_{min}(N) = P_{mit}(N) = P_{max}(N) = \left\lceil \log\left(\frac{N}{I}\right) \right\rceil$$

bei vier Bändern und Internspeichergröße $I$.

## 2.7.3 Ausgeglichenes Mehr-Wege-Mergesort

Das im Abschnitt 2.7.2 beschriebene Verfahren des 2-Wege-Mergesort lässt sich leicht auf ein Mehr-Wege-Verschmelzen verallgemeinern.

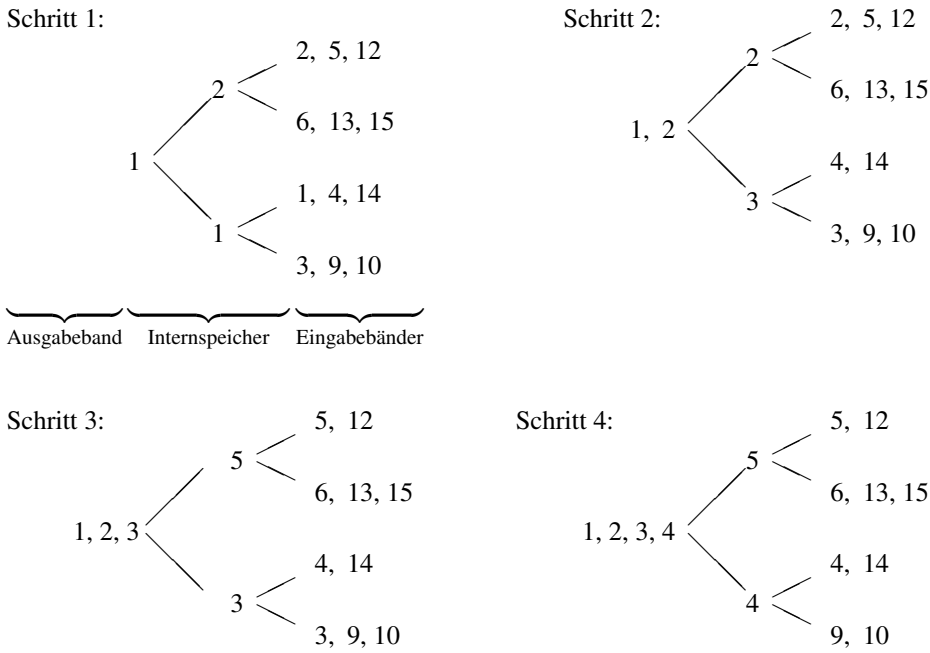
**Methode:** Wir verwenden $2k$ Bänder, $t_1, \ldots, t_{2k}$. Das Eingabeband ist $t_1$. Es werden wiederholt $I$ Datensätze von $t_1$ gelesen, intern sortiert und abwechselnd auf $t_{k+1}, t_{k+2}, \ldots, t_{2k}$ geschrieben solange, bis $t_1$ erschöpft ist. Dann stehen etwa $\frac{N}{(k \cdot I)}$ Runs der Länge $I$ auf $t_i$, $k+1 \leq i \leq 2k$. Die $k$ Bänder $t_{k+1}, \ldots, t_{2k}$ sind jetzt die Eingabebänder für ein $k$-Wege-Verschmelzen, die $k$ Bänder $t_1, \ldots, t_k$ sind die Ausgabebänder. Nun

werden die ersten Runs der Eingabebänder zu einem Run der Länge $k \cdot I$ verschmolzen und auf das Ausgabeband $t_1$ geschrieben. Dann werden die nächsten $k$ Runs der Eingabebänder verschmolzen und nach $t_2$ geschrieben. So werden der Reihe nach Runs der Länge $k \cdot I$ auf die Ausgabebänder geschrieben, bis die Eingabebänder erschöpft sind. Nach dieser Verschmelzungs- und Aufteilungsphase tauschen die Eingabe- und Ausgabebänder ihre Rollen. Das $k$-Wege-Verschmelzen und $k$-Wege-Aufteilen wird solange fortgesetzt, bis die gesamte Folge der Datensätze als ein Run auf einem der Bänder steht.

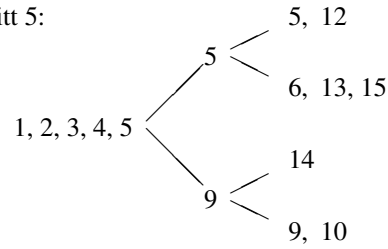
Beim $k$-Wege-Verschmelzen von Runs auf $k$ Bändern wird zunächst der erste Datensatz jedes Runs in den Internspeicher gelesen. Von den $k$ intern gespeicherten Datensätzen wird derjenige mit kleinstem Schlüssel für die Ausgabe ausgewählt und auf das Ausgabeband geschrieben. Der ausgewählte Datensatz wird dann im Internspeicher ersetzt durch den nächsten Datensatz im zugehörigen Eingabe-Run; dieser wird vom entsprechenden Eingabeband gelesen. Dies wird solange fortgesetzt, bis alle $k$ Eingabe-Runs erschöpft sind.

Damit das Verschmelzen von $k$ Datensätzen im Internspeicher geschehen kann, muss $k \leq I$ gelten. Für kleine Werte von $k$ mag es vernünftig sein unter den gerade betrachteten $k$ Datensätzen denjenigen mit minimalem Schlüssel durch lineares Durchsehen aller $k$ Schlüssel zu bestimmen; für größere Werte von $k$ ist es vorteilhaft das $k$-Wege-Verschmelzen mittels einer Halde (Heap) oder eines *Auswahlbaumes* (*selection tree*) zu unterstützen. In einer solchen Datenstruktur kostet das Entfernen des Minimums und Hinzufügen eines neuen Schlüssels nur $O(\log k)$ Schritte (interne Vergleichs- und Rechenoperationen), wenn $k$ Schlüssel gespeichert sind (vgl. Abschnitt 2.3 und Kapitel 6).

Betrachten wir als Beispiel das 4-Wege-Verschmelzen mit einem 2-stufigen Auswahlbaum:

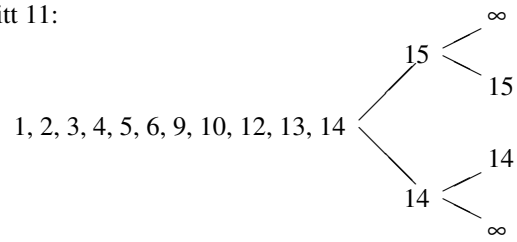


Schritt 5:

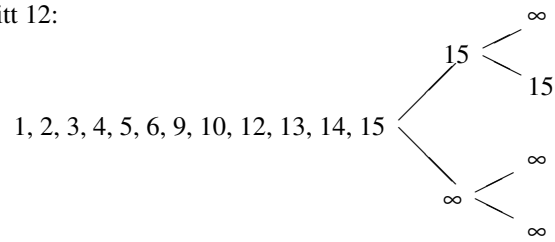



⋮

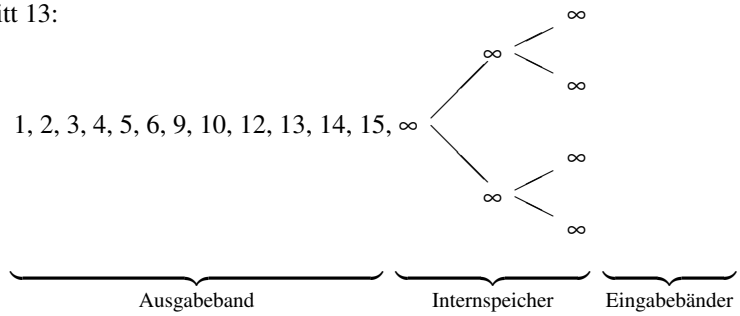
Schritt 11:




Schritt 12:




Schritt 13:




Diese Methode, genannt *Auswahl und Ersetzen* (*replacement selection*), lässt sich vorteilhaft auch schon für die Anfangsverteilung verwenden, indem man die Eingabefolge mit sich selbst $k$-Wege-verschmilzt.

**Auswahl und Ersetzen für eine unsortierte Eingabefolge:** Die Datensätze stehen unsortiert auf dem Eingabeband. Der Internspeicher fasst $I$ Datensätze. Zunächst werden die ersten $I$ Datensätze vom Eingabeband gelesen. Von den $I$ intern gespeicherten Datensätzen wird derjenige mit kleinstem Schlüssel für die Ausgabe ausgewählt und auf das Ausgabeband geschrieben. Der ausgewählte Datensatz wird im Internspeicher ersetzt durch den nächsten Datensatz auf dem Eingabeband. Ist der neue Schlüssel nicht kleiner als der des soeben ausgegebenen Datensatzes, dann wird der neue Datensatz noch zum gleichen Run gehören wie der soeben ausgegebene. Ist er jedoch kleiner, so gehört er zum nächsten Run; er wird erst ausgegeben, wenn der aktuelle Run beendet ist. Als nächster wird der kleinste Schlüssel ausgewählt, der nicht kleiner ist als der soeben ausgegebene; er kann noch zum aktuellen Run hinzugefügt werden. Dies wird solange wiederholt, bis alle Schlüssel im Internspeicher kleiner sind als der zuletzt ausgegebene; dann muss ein neuer Run angefangen werden.

**Beispiel:** Betrachten wir die Folge $F = 12,5,2,15,13,6,14,1,4,9,10,3,11,7,8$ aus Abschnitt 2.7.2, für $I = 3$:

Anfangs: $\underbrace{\quad}_{\text{Ausgabeband}} \mid \underbrace{12,5,2}_{\text{Internspeicher}} \mid \underbrace{15,13,6,14,1,4,9,10,3,11,7,8}_{\text{Eingabeband}}$

Schritt 1: 2 | 12, 5, 15 | 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 2: 2, 5 | 12, 15, 13 | 6, 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 3: 2, 5, 12 | 15, 13, 6 | 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 4: 2, 5, 12, 13 | 15, 6, 14 | 1, 4, 9, 10, 3, 11, 7, 8
Schritt 5: 2, 5, 12, 13, 14 | 15, 6, 1 | 4, 9, 10, 3, 11, 7, 8
Schritt 6: 2, 5, 12, 13, 14, 15 | 6, 1, 4 | 9, 10, 3, 11, 7, 8

Alle Schlüssel im Internspeicher sind kleiner als der zuletzt ausgegebene; daher wird ein neuer Run angefangen.

Schritt 7: 2, 5, 12, 13, 14, 15; 1 | 6, 4, 9 | 10, 3, 11, 7, 8
Schritt 8: 2, 5, 12, 13, 14, 15; 1, 4 | 6, 9, 10 | 3, 11, 7, 8
⋮
Schritt 15: 2, 5, 12, 13, 14, 15; 1, 4, 6, 9, 10, 11; 3, 7, 8

Am Beispiel erkennt man, dass die Länge von Runs $I$ übersteigen kann. Man kann zeigen, dass beim Verfahren Auswahl und Ersetzen die durchschnittliche Länge von Runs $2 \cdot I$ ist, Runs also im Mittel doppelt so lang sind wie beim internen Sortieren nach Abschnitt 2.7.2 (vgl. [100]). Außerdem werden Runs, die schon in der Eingabefolge vorhanden sind, noch verlängert, also eine Vorsortierung berücksichtigt.

**Analyse:** Anfangs werden mittels Auswahl und Ersetzen mindestens ein Run, höchstens $\lceil \frac{N}{I} \rceil$ Runs und im Mittel $\lceil \frac{N}{(2 \cdot I)} \rceil$ Runs hergestellt. Nach jeder Verschmelzungs- und Verteilungsphase hat sich die Anzahl der Runs auf das $1/k$-fache verringert. Damit ist

$$P_{min}(N) = 1; \quad P_{mit}(N) = \left\lceil \log_k \left( \frac{N}{(2 \cdot I)} \right) \right\rceil; \quad P_{max}(N) = \left\lceil \log_k \left( \frac{N}{I} \right) \right\rceil$$

bei $2k$ Bändern und Internspeichergröße $I$.

## 2.7.4 Mehrphasen-Mergesort

Während beim ausgeglichenen $k$-Wege-Mergesort alle $k$ Eingabebänder benutzt werden, wird nur auf eines der Ausgabebänder geschrieben. Die anderen Ausgabebänder sind temporär nutzlos. Da normalerweise $k$ viel kleiner als $I$ ist, wäre es wünschenswert diese Bänder mit als Eingabebänder heranzuziehen. Man will aber sicherlich nicht alle Runs auf ein einziges Ausgabeband speichern, weil man sonst vor der nachfolgenden Verschmelze-Phase noch eine zusätzliche Verteilungsphase einschieben müsste.

**Methode:** Beim *Mehrphasen-Mergesort* (*polyphase mergesort*) arbeitet man mit $k+1$ Bändern, von denen zu jedem Zeitpunkt $k$ Eingabebänder sind und eines das Ausgabeband ist. Man schreibt solange alle entstehenden Runs auf das Ausgabeband, bis eines der Eingabebänder erschöpft ist (das ist eine *Phase*). Dann wird das leer gewordene Eingabeband zum Ausgabeband und das bisherige Ausgabeband wird zurückgespult und dient als ein Eingabeband. Damit hat man wieder $k$ Eingabebänder und ein (anderes) Ausgabeband. Der Sortiervorgang ist beendet, wenn alle Eingabebänder erschöpft sind.

Dieses Verfahren funktioniert nur dann wie beabsichtigt, wenn zu jedem Zeitpunkt (außer am Schluss) nur ein Eingabeband leer wird. Betrachten wir ein Beispiel für drei Bänder. In der Anfangsverteilung seien die 13 Runs $r_1, r_2, \ldots, r_{13}$ auf die beiden Bänder $t_1$ und $t_2$ verteilt wie folgt ($t_3$ ist leer):

| $t_1$ | $t_2$ | $t_3$ |
|---|---|---|
| $r_1 r_2 \ldots r_8$ | $r_9 r_{10} \ldots r_{13}$ | leer |

Dann werden die jeweils nächsten Runs von $t_1$ und $t_2$ verschmolzen und auf das Ausgabeband $t_3$ geschrieben, bis $t_2$ erschöpft ist (das ist die erste Phase).

$$r_6 r_7 r_8 \;\Big|\; \text{leer} \;\Big|\; r_{1,9} r_{2,10} r_{3,11} r_{4,12} r_{5,13}$$

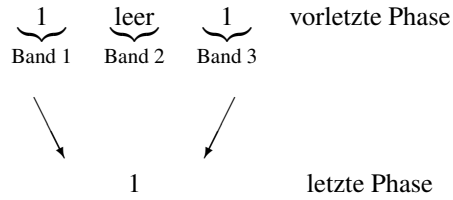
Jetzt wird $t_2$ zum Ausgabeband; die jeweils nächsten Runs von $t_1$ und $t_3$ werden verschmolzen, bis $t_1$ leer ist.

$$\text{leer} \;\Big|\; r_{6,1,9} r_{7,2,10} r_{8,3,11} \;\Big|\; r_{4,12} r_{5,13}$$

Wir zeigen die Situation nach jeder Phase, bis zum Ende:

| | | |
|---|---|---|
| $r_{6,1,9,4,12} r_{7,2,10,5,13}$ | $r_{8,3,11}$ | leer |
| $r_{7,2,10,5,13}$ | leer | $r_{6,1,9,4,12,8,3,11}$ |
| leer | $r_{7,2,10,5,13,6,1,9,4,12,8,3,11}$ | leer |

Der Aufwand für eine Phase ist dabei im Allgemeinen (außer für die Anfangsverteilung und für die letzte Phase) niedriger als für einen Durchgang beim Mehrwege-Mergesort, weil in einer Phase nicht alle Datensätze bearbeitet werden. Überlegen wir uns anhand der obigen Illustration, wie denn die Runs anfangs auf die beiden Eingabebänder verteilt sein müssen, damit nach jeder Phase (außer der letzten) nur ein Band erschöpft ist. Am Ende muss gerade ein Run auf einem Band stehen. Dieser Run muss entstehen aus zwei Runs, die auf zwei Bändern stehen.




Einer der beiden Runs der vorletzten Phase, sagen wir ohne Einschränkung der Allgemeinheit der Run auf Band 1, muss entstanden sein aus einem Run auf Band 2, das leer wurde, und einem Run auf Band 3:

| Band 1 | Band 2 | Band 3 | Phase |
|---|---|---|---|
| | 1 | | $i$ |
| 1 | | 1 | $i-1$ |
| | 1 | 2 | $i-2$ |

Entsprechend können wir Situationen in vorangehenden Phasen konstruieren, die zum gewünschten Ergebnis führen.

| | | |
|---|---|---|
| | 1 | |
| 1 | | 1 |
| | 1 | 2 |
| 2 | 3 | |
| 5 | | 3 |
| | 5 | 8 |
| 8 | 13 | |
| | $\vdots$ | |
| $F_{n-1}$ | $F_n$ | |
| $F_{n+1} = F_{n-1} + F_n$ | | $F_n$ |

Damit zeigt sich, dass die Run-Zahlen auf den beiden Anfangsbändern von der Form $F_{n-1}$ und $F_n$ sein müssen, mit

$$F_0 = 0,\ F_1 = 1 \text{ und } F_n = F_{n-1} + F_{n-2},\ n \geq 2.$$

Das sind gerade die Fibonacci-Zahlen. Man kann eine Anfangsverteilung auf zwei der drei Bänder so, dass die Run-Zahlen gerade zwei aufeinander folgende Fibonacci-Zahlen sind, erreichen indem man fiktive Runs nach Bedarf zu den durch die Eingabe erhaltenen Runs hinzunimmt (natürlich ohne sie wirklich abzuspeichern).

**Analyse:** Man kann zeigen (vgl. [100]), dass Mehrphasen-Mergesort bei drei Bändern im Mittel

$$P_{mit}(N) = 1.04 \cdot \log S + 0.99$$

Durchgänge benötigt, wenn man anfangs aus den $N$ Datensätzen $S$ Runs erzeugt. Damit kann man mit Mehrphasen-Mergesort und drei Bändern etwa gleich schnell sortieren wie mit ausgeglichenem Mehrwege-Mergesort und vier Bändern.

Die Strategie des Mehrphasen-Mergesort lässt sich auch auf mehr als drei Bänder anwenden. Die Verteilung der Runs auf die $k$ Eingabebänder muss dann den Fibonacci-Zahlen höherer Ordnung folgen:

$$\begin{aligned} F_n^k &= F_{n-1}^k + F_{n-2}^k + \cdots + F_{n-k}^k \quad \text{für} \quad n \geq k, \quad \text{und} \\ F_n^k &= 0 \quad \text{für} \quad 0 \leq n \leq k-2, \quad \text{und} \\ F_{k-1}^k &= 1. \end{aligned}$$

Die Anzahl der zum Sortieren von $N$ Datensätzen mit $k+1$ Bändern benötigten Durchgänge ist dann für größere Werte von $k$ (ab $k = 6$) etwa

$$P_{mit}(N) = 0.5 \log S,$$

wenn man anfangs $S$ Runs erzeugt hat.

Weder haben wir in diesem Abschnitt alle grundlegenden Strategien zum externen Sortieren mit Bändern behandelt noch sind externe Mergesort-Verfahren erschöpfend betrachtet worden. Der interessierte Leser findet in [100] noch eine Fülle von Verfahren und Überlegungen, auch praktischer Natur. Beispielsweise kann man versuchen auch die Rückspulzeit des Ausgabebandes beim Phasenende des Mehrphasen-Mergesort zum Sortieren zu nutzen; diesem Gedanken folgt das *kaskadierende Mergesort* (*cascade mergesort*). Wenn das Magnetband auch rückwärts gelesen werden kann und zwischen Lesen und Schreiben vorwärts und rückwärts umgeschaltet werden kann, so lässt sich das *oszillierende Mergesort* (*oscillating mergesort*) einsetzen. Als Zusammenfassung vieler Überlegungen formuliert D. Knuth einen Satz [100]:

**Theorem** *It is difficult to decide which merge pattern is best in a given situation.*

Dieser Satz bedarf sicherlich keines Beweises.

## 2.8 Untere Schranken

Die große Zahl und unterschiedliche Laufzeit der von uns diskutierten Sortierverfahren legt die Frage nahe, wie viele Schritte zum Sortieren von $N$ Datensätzen denn mindestens benötigt werden. In dieser Form ist die Frage natürlich zu vage formuliert um eine präzise Antwort zu ermöglichen. Der zum Sortieren von $N$ Datensätzen mindestens erforderliche Aufwand hängt ja unter anderem davon ab, was wir über die zu sortierenden Datensätze wissen und welche Operationen wir im Sortierverfahren zulassen. Wir wollen daher zunächst nur die Klasse der allgemeinen Sortierverfahren betrachten. Das sind Verfahren, die zur Lösung des Sortierproblems nur Vergleichsoperationen zwischen Schlüsseln benutzen. Alle von uns vorgestellten allgemeinen Sortierverfahren haben jedenfalls im schlechtesten Fall wenigstens $\Omega(N \log_2 N)$ Vergleichsoperationen zwischen Schlüsseln benötigt. Diese Zahl von Vergleichsoperationen war auch im Mittel erforderlich, wenn man über alle $N!$ möglichen Anordnungen von $N$ Schlüsseln mittelt und jede Anordnung als gleich wahrscheinlich ansieht. Kann man mit weniger

Vergleichsoperationen auskommen? Die Antwort ist „ja“ , wenn man etwa weiß, dass die Datensätze gut vorsortiert sind, vgl. hierzu Abschnitt 2.6, aber „nein“ im schlechtesten Fall und im Mittel. Wir zeigen dazu den folgenden Satz.

**Satz 2.4** *Jedes allgemeine Sortierverfahren benötigt zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel wenigstens* $\Omega(N \log N)$ *Schlüsselvergleiche.*

Zum Beweis dieses Satzes müssen wir uns eine Übersicht über alle möglichen allgemeinen Sortierverfahren verschaffen. Wir haben das bereits in Abschnitt 2.6 mithilfe von Entscheidungsbäumen getan. Im Entscheidungsbaum kann man die von einem allgemeinen Sortierverfahren ausgeführten Schlüsselvergleiche fest halten. Jeder innere Knoten enthält ein Paar $(i, j)$ von Indizes und repräsentiert einen Vergleich zwischen den Schlüsseln $k_i$ und $k_j$, den ein Sortierverfahren $A$ für $N$ Schlüssel $k_1, \ldots, k_N$ ausführt. Am Anfang hat $A$ keine Information über die „richtige“ Anordnung von $k_1, \ldots, k_N$, d. h. alle $N!$ Permutationen sind möglich. Nach einem Vergleich der Schlüssel $k_i$ und $k_j$ kann $A$ jeweils alle diejenigen Permutationen ausschließen, in denen $k_i$ vor bzw. hinter $k_j$ auftritt. Das wird im Entscheidungsbaum durch einen mit „$i : j$“ beschrifteten Knoten mit zwei Ausgängen modelliert, dessen linker Ausgang die Bedingung „$k_i \leq k_j$“ und dessen rechter Ausgang die Bedingung „$k_i > k_j$“ repräsentiert. Jedes Blatt ist mit derjenigen Permutation der Schlüssel $k_1, \ldots, k_N$ markiert, die alle Bedingungen erfüllt, die auf dem Weg von der Wurzel zu diesem Blatt auftreten. (Ein Beispiel für einen Entscheidungsbaum mit vier Schlüsseln zeigt Abbildung 2.10). Die zum Sortieren von $k_1 \ldots, k_N$ durch $A$ ausgeführte Anzahl von Schlüsselvergleichen ist die Zahl der Knoten auf dem Pfad von der Wurzel des Entscheidungsbaumes zum mit $(k_1, \ldots, k_N)$ markierten Blatt. Man nennt diese Zahl die *Tiefe* des Blattes. (Vgl. hierzu auch das Kapitel 5.) Der Algorithmus $A$ könnte natürlich überflüssige Vergleiche durchführen um die Permutation $k_1, \ldots, k_N$ zu identifizieren. Weil wir aber an der Mindestanzahl von Vergleichsoperationen interessiert sind, können wir annehmen, dass keine überflüssigen Vergleiche auftreten. Weil alle $N!$ Anordnungen der $N$ Schlüssel möglich sind, muss der das Verfahren $A$ modellierende Entscheidungsbaum (mindestens) $N!$ Blätter haben. Zum Nachweis der unteren Schranke genügt es nun die folgende Behauptung zu zeigen:

**Behauptung** *Die maximale und die mittlere Tiefe eines Blattes in einem Binärbaum mit k Blättern ist wenigstens* $\log_2 k$.

Der erste Teil der Behauptung ist trivial, weil ein Binärbaum, dessen sämtliche Blätter höchstens die Tiefe $t$ haben, auch nur höchstens $2^t$ Blätter haben kann. Zum Nachweis des zweiten Teils nehmen wir an, die Behauptung sei falsch. Sei $T$ der kleinste Binärbaum, für den die Behauptung nicht gilt. $T$ habe $k$ Blätter. Dann muss $k \geq 2$ sein und $T$ einen linken Teilbaum $T_1$ mit $k_1$ Blättern und einen rechten Teilbaum $T_2$ mit $k_2$ Blättern haben, mit $k_1 < k$, $k_2 < k$ und $k_1 + k_2 = k$ (siehe Abbildung 2.14). Da $T_1$ und $T_2$ kleiner sind als $T$, muss gelten:

$$\text{mittlere Tiefe } (T_1) \geq \log_2 k_1$$

$$\text{mittlere Tiefe } (T_2) \geq \log_2 k_2.$$

Offenbar ist für jedes Blatt von $T$ die Tiefe, bezogen auf die Wurzel von $T$, um genau eins größer als die Tiefe des Blattes in $T_1$ bzw. $T_2$. Daher gilt:

$$\begin{aligned}\text{mittlere Tiefe } (T) &= \frac{k_1}{k}(\text{ mittlere Tiefe } (T_1)+1)+\frac{k_2}{k}(\text{ mittlere Tiefe } (T_2)+1)\\ &\geq \frac{k_1}{k}(\log_2 k_1+1)+\frac{k_2}{k}(\log_2 k_2+1)\\ &= \frac{1}{k}(k_1\log_2(2k_1)+k_2\log_2(2k_2)) = f(k_1,k_2)\end{aligned}$$

Diese Funktion $f(k_1,k_2)$ nimmt ihr Minimum unter der Nebenbedingung $k_1+k_2=k$ für $k_1=k_2=\frac{k}{2}$ an; also ist

$$\text{mittlere Tiefe } (T) \geq \frac{1}{k}(\frac{k}{2}\log_2 k+\frac{k}{2}\log_2 k) = \log_2 k.$$

Die Behauptung gilt also doch für $T$ im Widerspruch zur Annahme.

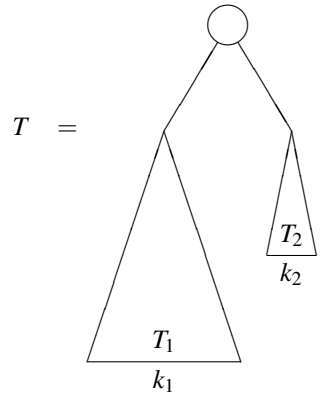



**Abbildung 2.14**

Aus der soeben bewiesenen Behauptung kann man nun die untere Schranke für die maximale und mittlere Zahl von Vergleichsoperationen zum Sortieren von $N$ Schlüsseln mithilfe eines allgemeinen Sortierverfahrens leicht herleiten. Da der zugehörige Entscheidungsbaum wenigstens $N!$ Blätter haben muss, ist die maximale und mittlere Tiefe eines Blattes wenigstens $\log(N!) \in \Omega(N\log N)$. Denn es ist $N! \geq (\frac{N}{2})^{\frac{N}{2}}$ und damit $\log(N!) \geq \frac{N}{2}\log(\frac{N}{2}) = \Omega(N\log N)$. □

Kann man $N$ Zahlen im Mittel und im schlechtesten Fall schneller als in größenordnungsmäßig $N\log N$ Schritten sortieren, wenn man andere Operationen, also nicht nur Vergleichsoperationen zwischen Schlüsseln zulässt? Ein bemerkenswertes Ergebnis dieser Richtung wurde 1978 von W. Dobosiewicz [43] erzielt. Er hat ein Sortierverfahren angegeben, das neben arithmetischen Operationen auch noch die so genannte Floor-Funktion „$\lfloor\ \rfloor$“ benutzt, die einer reellen Zahl $x$ die größte ganze Zahl $i \leq x$ zuordnet,

und gezeigt: Das Verfahren erlaubt es $N$ reelle Zahlen stets in $O(N \log N)$ Schritten zu sortieren; der Erwartungswert der Laufzeit des Verfahrens für $N$ gleich verteilte Schlüssel ist sogar $O(N)$.

Wir zeigen jetzt, dass es nichts nützt, wenn man nur die arithmetischen Operationen $+, -, *, /$ zulässt. Wir lassen also die Annahme fallen, dass Vergleichsoperationen die einzigen zugelassenen Operationen zwischen Schlüsseln sind. Stattdessen nehmen wir an, die zu sortierenden Schlüssel seien reelle Zahlen, die addiert, subtrahiert, multipliziert und dividiert werden dürfen. Wir verallgemeinern das Konzept von Entscheidungsbäumen wie folgt zum Konzept von rationalen Entscheidungsbäumen.

Ein *rationaler Entscheidungsbaum* für $N$ reellwertige Eingabevariablen ist ein Binärbaum, dessen innere Knoten Entscheidungen abhängig vom Wert rationaler Funktionen für die Eingabevariablen repräsentieren. Jedem inneren Knoten $i$ ist eine Funktion $B_i(x_1, \ldots, x_N)$ zugeordnet; bei $i$ wird nach links verzweigt, falls $B_i(x_1, \ldots, x_N) > 0$ (oder: $B_i(x_1, \ldots, x_N) \geq 0$) ist, und nach rechts sonst. Ferner wird jedem Blatt $j$ eine rationale Funktion $A_j(x_1, \ldots, x_N)$ zugeordnet.

Ein rationaler Entscheidungsbaum berechnet in offensichtlicher Weise eine auf einer Menge $W \subseteq \mathbb{R}^N$ definierte, reellwertige Funktion. Betrachten wir dazu das in Abbildung 2.15 gezeigte Beispiel.

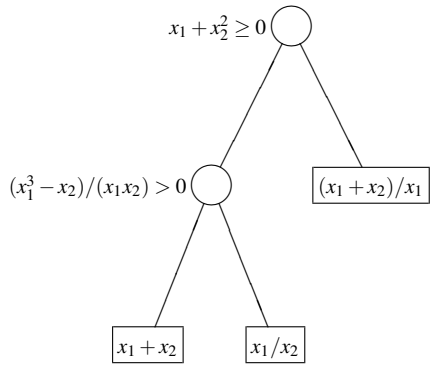



**Abbildung 2.15**

Dieser rationale Entscheidungsbaum berechnet folgende, reellwertige Funktion:

$$f(x_1,x_2) = \begin{cases} x_1 + x_2; & \text{falls} \quad x_1 + x_2^2 \geq 0 \quad \text{und} \quad (x_1^3 - x_2)/(x_1 x_2) > 0 \\ x_1/x_2; & \text{falls} \quad x_1 + x_2^2 \geq 0 \quad \text{und} \quad (x_1^3 - x_2)/(x_1 x_2) < 0 \\ (x_1 + x_2)/x_1; & \text{falls} \quad x_1 + x_2^2 < 0 \end{cases}$$

Diese Funktion $f$ ist auf dem Gebiet $W \subseteq \mathbb{R}^2$ definiert, dessen Gestalt und Eigenschaften uns hier nicht interessieren.

Allgemein kann man die von einem rationalen Entscheidungsbaum berechnete Funktion $f$ von $N$ reellwertigen Variablen $X = x_1, \ldots, x_N$ schreiben in der Form:

$$f(X) = \begin{cases} A_1(X); & \text{falls} \quad X \in M_1 \\ \quad \vdots & \\ A_m(X); & \text{falls} \quad X \in M_m \end{cases}$$

Dabei sind die $A_j(X)$ die an den Blättern des Entscheidungsbaumes stehenden Funktionen und $M_j$ bezeichnet die Konjunktion der Bedingungen, die auf dem Pfad von der Wurzel bis zum mit $A_j$ beschrifteten Blatt gültig sind.

Wir stellen nun eine Verbindung her zwischen Sortierverfahren, die die Operationen $\{<, +, -, *, /\}$ benutzen dürfen, und rationalen Entscheidungsbäumen zur Berechnung von Funktionen. Wir betrachten dazu die so genannte *Sortierindexfunktion* von $N$ Argumenten $f(x_1, \ldots, x_N) = x_1^{r_1} + \cdots + x_N^{r_N}$, mit $r_i = Rang$ von $x_i$ in der Folge der Argumente bei aufsteigender Sortierung. Für $N = 2$ gilt also beispielsweise:

$$f(x_1, x_2) = \begin{cases} x_1^1 + x_2^2; & \text{falls} \quad x_1 < x_2 \\ x_1^2 + x_2^1; & \text{falls} \quad x_1 \geq x_2 \end{cases}$$

Kann man nun das Sortierproblem für $N$ reelle Zahlen $x_1, \ldots, x_N$ mit $k$ Vergleichsoperationen des Typs $B_i(x_1, \ldots, x_N) > 0$ oder $B_i(x_1, \ldots, x_N) \geq 0$ für rationale Funktionen $B_i$ lösen, so kann man auch die Sortierindexfunktion von $N$ Argumenten mit $k$ derartigen Vergleichsoperationen berechnen. Denn der Wert der Sortierindexfunktion kann ohne weitere Vergleichsoperation berechnet werden, sobald die Anordnung der Argumente bekannt ist. Damit liefert eine untere Schranke für die Berechnung der Sortierindexfunktion auch eine untere Schranke für das Sortierproblem. Eine untere Schranke für die Berechnung der Sortierindexfunktion kann man aber – wie im Falle „gewöhnlicher" Entscheidungsbäume – sofort aus einer unteren Schranke für die Blattzahl eines rationalen Entscheidungsbaums zur Berechnung der Sortierindexfunktion ableiten. Um eine solche Schranke herzuleiten, zeigen wir ganz allgemein, dass jeder rationale Entscheidungsbaum zur Berechnung einer Funktion $f : \mathbb{R}^N \to \mathbb{R}$, die in wenigstens $q$ verschiedene Teile zerfällt, wenigstens $q$ Blätter haben muss. Genauer zeigen wir (vgl. [179]):

**Satz 2.5** *Sei $f : \mathbb{R}^N \to \mathbb{R}$ eine auf $W \subseteq \mathbb{R}^N$ definierte Funktion, seien $X_1, \ldots, X_q \in W$ paarweise verschiedene Punkte und $Q_1, \ldots, Q_q$ paarweise verschiedene rationale Funktionen, die auf Kreisen mit Radius $e > 0$ um $X_1, \ldots, X_q$ definiert sind und dort mit $f$ übereinstimmen, also:*

$$f(X) = Q_i(X), \qquad \textit{für alle} \quad X \in U(X_i, e) = \{X : \mid X_i - X \mid \; < e\} \subseteq W.$$

*Dann muss jeder Entscheidungsbaum zur Berechnung von $f$ wenigstens $q$ Blätter haben.*

Zum Beweis dieses Satzes benutzt man wohl bekannte Fakten aus der algebraischen Geometrie. Wir skizzieren die Argumentationskette und verweisen auf [179] für weitere Einzelheiten.

Den durch einen rationalen Entscheidungsbaum zur Berechnung von $f$ gegebenen Algorithmus $A$ kann man schreiben in der Form:

$$A(X) = \begin{cases} A_1(X); & \text{falls} \quad X \in M_1 \\ \quad \vdots & \\ A_m(X); & \text{falls} \quad X \in M_m \end{cases}$$

Dabei ist $m$ die Anzahl der Blätter des Entscheidungsbaumes. Weil $A$ die Funktion $f$ berechnet, muss für jedes $i$ gelten:

$$U(X_i, e) = (U(X_i, e) \cap M_1) \cup \ldots \cup (U(X_i, e) \cap M_m)$$

Auf der rechten Seite dieser Gleichung steht die endliche Vereinigung von Mengen, die durch rationale Bedingungen definiert sind. Weil die linke Seite eine Menge ist, die alle Punkte einer ganzen Kreisscheibe mit positivem Radius enthält, muss auch wenigstens eine der Mengen $(U(X_i, e) \cap M_j)$ diese Eigenschaft haben! (Das ist das erste Ergebnis aus der algebraischen Geometrie, das wir benötigen.) Es gibt also ein $j$, sodass für alle Punkte einer ganzen Kreisscheibe in $M_j$ die Funktionen $Q_j$ und $A_j$ dort übereinstimmen. Weil $Q_j$ und $A_j$ rationale Funktionen sind, müssen sie dann überhaupt identisch sein. (Das ist das zweite benötigte Ergebnis aus der algebraischen Geometrie; man kann es als eine Verallgemeinerung des bekannten Nullstellensatzes für Polynome auffassen.) Also muss es wenigstens so viele verschiedene $A_j$ wie $Q_j$ geben, d. h. $m \geq q$. □

Kehren wir nun zur Sortierindexfunktion zurück und wenden wir Satz 2.5 darauf an. Diese Funktion ist für die $q = N!$ verschiedenen Punkte $X_\pi = (\pi(1), \ldots, \pi(N))$, $\pi$ Permutation von $\{1, \ldots, N\}$, definiert und stimmt jeweils auf einem Kreis mit Radius $e > 0$, $e < \frac{1}{2}$, um diese Punkte mit einer rationalen Funktion, der Funktion $Q_\pi(X_1, \ldots, X_N) = X_1^{\pi(1)} + \cdots + X_N^{\pi(N)}$, überein. Daher muss jeder rationale Entscheidungsbaum zur Berechnung der Sortierindexfunktion wenigstens $N!$ Blätter haben. Die maximale und mittlere Tiefe eines Blattes ist damit in $\Omega(N \log N)$.

In den letzten Jahren ist es gelungen wesentlich stärkere Sätze dieser Art zum Nachweis unterer Schranken zu beweisen. Dazu wurden so genannte *algebraische Entscheidungsbäume* definiert und Sätze analog zu Satz 2.5 für solche Bäume bewiesen. Zum Nachweis dieser Sätze werden aber mächtige Hilfsmittel aus der algebraischen Geometrie benötigt, die weit über den Rahmen dieses Buches hinausgehen. Den interessierten Leser verweisen wir auf [135] und auf die Originalarbeit [15].

## 2.9 Implementation und Test von Sortierverfahren in Java

Allgemeine Sortierverfahren sind dadurch charakterisiert, dass sie als einzige Information zur Anordnung der anzuordnenden Objekte Resultate von Schlüsselvergleichen verwenden. Das Verhalten von in diesem Sinne vergleichbaren Objekten kann in Java durch eine Schnittstelle *Orderable* wie folgt beschrieben werden:

```
public interface Orderable {
   public boolean equal (Orderable o);
   public boolean greater (Orderable o);
   public boolean greaterEqual (Orderable o);
   public boolean less (Orderable o);
   public boolean lessEqual (Orderable o);
   public Orderable minKey ();
}
```

Weil wir in der Regel ganzzahlige Schlüssel vorausgesetzt haben, können wir annehmen, dass die zu sortierenden Objekte einen ganzzahligen Schlüssel vom Typ *int* haben, der die Grundlage für den Vergleich von Objekten darstellt.

```
public class OrderableInt implements Orderable {
   protected int i;       // Schlüssel
   // evtl. weitere Komponenten mit „eigentlicher“ Information
   /* Hier folgt die Implementation aller Operationen der Schnittstelle
      Orderable */
}
```

Ein Rahmen zur Implementation und zum Test von Sortierverfahren initialisiert ein Array von Objekten der Klasse *OrderableInt* mit den zu sortierenden Schlüsseln, ruft das jeweilige Sortierverfahren auf und sorgt für die Ausgabe der im Array gespeicherten Werte vor und nach der Sortierung. Dabei wird angenommen, dass an der Array-Position 0 kein zu sortierendes Objekt, sondern ein so genannter „Stopper“ gespeichert ist; die $n$ zu sortierenden Werte stehen also an den Positionen $1, \ldots, n$.

Alle allgemeinen Sortierverfahren können dann als Unterklasse einer wie folgt definierten Sortierbasisklasse implementiert werden:

```
abstract public class SortAlgorithm {
   static void swap (Object A[], int i, int j) {
      Object o = A[i]; A[i] = A[j]; A[j] = o;
   }
   static void sort (Orderable A[]) {}
   static void printArray (Orderable A[]) {
      for (int i = 1; i<A.length; i++)
         System.out.print(A[i].toString()+" ");
      System.out.println();
   };
}
```

Als Beispiel für die Implementation eines allgemeinen Sortierverfahrens in Java geben wir hier nur die Implementation des Sortierens durch Einfügen an:

```
public class EinfuegeSort extends SortAlgorithm {
   static void sort (Orderable A[]) {
```

```
    for (int i = 2; i < A.length; i++) {
      Orderable temp = A[i];
      int j = i − 1;
      while (j >= 1 && A[j].greater(temp)) {
        A[j+1] = A[j];
        j−−;
      }
      A[j+1] = temp;
    }
  }
}
```

In analoger Weise können die Verfahren Bubblesort, Shellsort, Sortieren durch Auswahl, Quicksort und Heapsort implementiert werden, also sämtliche Verfahren, die *am Ort* d. h. ohne zusätzlichen, von der Anzahl der zu sortierenden Objekte (linear) abhängenden Speicherplatz operieren.

Das durch rekursives Verschmelzen von sortierten Teilfolgen arbeitende Verfahren Mergesort benötigt zum Verschmelzen bereits sortierter Folgen ein zusätzliches Array, dessen Länge der Summe der Längen der zwei zu verschmelzenden Teilfolgen ist. Man spezifiziert also zunächst eine Methode *merge*, die die sortierten Folgen $A[l \,..\, m]$ und $A[m+1 \,..\, r]$ zu einer Folge $A[l \,..\, r]$ verschmilzt:

```
static void merge (comparable A[], int l, int m, int r) {
  comparable B [] = new comparable [A.length];
  int i = l;                    // Zeiger in A[l],...,A[m]
  int j = m + 1;                // Zeiger in A[m+1],...,A[r]
  int k = l;                    // Zeiger in B[l],...,B[r]
  while (i <= m && j <= r) {
    if (A[i].less(A[j])) {
      B[k] = A[i]; i++;
    }
    else {
      B[k] = A[j]; j++;
    }
    k++;
  }
  if (i > m) {                  // erste Teilfolge erschöpft, übernimm zweite
    for (int h = j; h <= r; h++, k++) {
      B[k] = A[h];
    }
  }
  else {                        // zweite Teilfolge erschöpft, übernimm erste
    for (int h = i; h <= m; h++, k++) {
      B[k] = A[h];
    }
  }
  for (int h = l; h <= r; h++) {
```

```
        A[h] = B[h];
    }
}
```

Dann kann das Verfahren *mergeSort* wie folgt implementiert werden:

```
class mergeSort {
    /* sortiert das ganze Array */
    public static void sort (comparable A[]) {
        sort (A, 1, A.length−1);
    }
    static void merge ...

    static void sort (comparable A[], int l, int r) {
        if (r > l) {
            int m = (l + r) / 2;
            sort(A, l, m);
            sort(A, m+1, r);
            merge(A, l, m, r);
        }
    }
}
```

Radix-exchange-sort und Sortieren durch Fachverteilung nutzen die spezielle Struktur der für die Sortierung maßgeblichen Schlüssel aus. Man kann auf das $i$-te Bit von Binärzahlen, auf die $i$-te Ziffer von Dezimalzahlen oder auf den $i$-ten Buchstaben von alphabetischen, also durch Zeichenreihen gegebenen Schlüsseln zugreifen. Diese Zerlegbarkeit der Schlüssel kann man in einer abstrakten Klasse *decomposable* mit Unterklassen *decomposableInt* für ganze Dualzahlen und *decomposableString* für Zeichenketten beschreiben.

Dann kann man beispielsweise die für das Verfahren Radix-exchange-sort typische Prüfung, ob das Bit an Position $i$ eines Schlüssels eines Objekts der Klasse *decomposableInt* gleich 0 oder 1 ist, durch eine Methode beschrieben werden, die eben dieses Bit zurückliefert:

```
public class decomposableInt extends decomposable {
    protected int i;        // Schlüsselkomponente ganzzahlig
      ⋮
    public int bit (int stelle) {
    // liefert das an stelle befindliche Bit der Schlüsselkomponente
    mask=1 << stelle;
    if ((mask & this.intValue()) == 0) return 0;
    return 1;
    }
}
```

Für die Implementation des Verfahrens Radix-exchange-sort muss dann natürlich vorausgesetzt werden, dass ein Array von Objekten des Typs *decomposableInt* gegeben ist. Das Verfahren Sortieren durch Fachverteilung kann entsprechend implementiert werden.

## 2.10 Aufgaben

**Aufgabe 2.1**

Sortieren Sie die unten angegebene, in einem Feld *a* gespeicherte Schlüsselfolge mit dem jeweils angegebenen Verfahren und geben Sie jede neue Belegung des Feldes nach einem Schlüsseltausch an.

a) 40–15–31–8–26–22; Auswahlsort

b) 35–22–10–51–48; Bubblesort.

**Aufgabe 2.2**

Schreiben Sie eine Pascal-Prozedur für Sortieren durch Auswahl, wobei die zu sortierende Zahlenfolge nicht in einem Array, sondern in Form einer linearen Liste vorliegt. Die Listenelemente seien durch folgende Typvereinbarung gegeben:

```
type listenzeiger = ↑listenelement;
     listenelement = record
                       key : integer;
                       next : listenzeiger
                     end;
```

Der Beginn der Liste werde durch ein Dummy-Element, das keinen Eintrag enthält und auf das ein Zeiger *head* zeigt, markiert. Das Listenende wird durch ein Listenelement gekennzeichnet, dessen *next*-Komponente den Wert **nil** hat. Die Liste enthalte mindestens zwei Elemente außer dem Dummy-Element.

**Aufgabe 2.3**

Geben Sie für die unten angegebenen Zahlenfolgen jeweils mit Begründung die Laufzeit der Sortierverfahren Auswahlsort, Einfügesort und Bubblesort in Groß-Oh-Notation an.

a) $1, \frac{N}{2}+1, 2, \frac{N}{2}+2, \ldots, \frac{N}{2}, N$ ($N$ gerade)

b) $N, 1, N-1, 2, N-2, 3, \ldots, N-\frac{N}{2}+1, \frac{N}{2}$ ($N$ gerade)

c) $N, 1, 2, 3, \ldots, N-1$

d) $2, 3, 4, \ldots, N, 1$

**Aufgabe 2.4**

Sei $N$ eine gerade Zahl. Wie groß ist der Aufwand zum Sortieren der Folge

$$\frac{N}{2}, \frac{N}{2}+1, \ldots, N, 1, 2, \ldots, \frac{N}{2}-1$$

bei den Sortierverfahren: 2-Wege-Mergesort, reines 2-Wege-Mergesort, natürliches 2-Wege-Mergesort?

**Aufgabe 2.5**

Gegeben sei das Array $a$ von neun Elementen mit den Schlüsseln

$$41 \quad 62 \quad 13 \quad 84 \quad 35 \quad 96 \quad 57 \quad 28 \quad 79.$$

Geben Sie alle Aufrufe der Prozedur *quicksort* und die Reihenfolge ihrer Abarbeitung an, die als Folge eines Aufrufs von $quicksort(a, 1, 9)$ im Hauptprogramm für obiges Array auftreten.

**Aufgabe 2.6**

Schreiben Sie in Pascal eine iterative Quicksort-Prozedur. D. h. die Quicksort-Prozedur wird nicht rekursiv für die neu entstandenen Teilfolgen aufgerufen, sondern man merkt sich die Grenzen der Teilfolgen, z. B. mithilfe eines Stapels. Sie dürfen die für Stapel üblichen Operationen verwenden ohne sie näher auszuführen. Achten Sie darauf, dass möglichst wenig zusätzlicher Speicherplatz benötigt wird.

**Aufgabe 2.7**

a) Gegeben sei die Schlüsselfolge $85, 20, 63, 18, 51, 37, 90, 33$. Erzeugen Sie für diese Folge einen Heap und stellen Sie ihn als Binärbaum dar.

b) Sortieren Sie die Schlüsselfolge $40, 15, 31, 8, 2, 6, 22$ (aufsteigend) mit Heapsort. Stellen Sie zunächst einen Heap her und geben Sie dann jede neue Belegung nach dem Schlüsseltausch an.

c) Geben Sie größenordnungsmäßig die Komplexität der folgenden Operationen auf einem Heap mit $N$ Elementen im schlimmsten Fall an. Am Ende einer jeden Operation soll stets ein Heap zurückbleiben.

- Einfügen eines beliebigen Elementes,
- Suchen des Maximums,
- Suchen eines beliebigen Elementes,
- Entfernen eines beliebigen Elementes,
- Suchen des Minimums,
- Entfernen des Maximums.

**Aufgabe 2.8**

Gegeben sei die in einem Array $a$ der Länge 10 abgelegte Schlüsselfolge

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $a$ : | 20 | 14 | 15 | 8 | 10 | 12 | 9 | 5 | 3 | 6 |

Sortieren Sie diese Folge in aufsteigender Reihenfolge mit dem Verfahren Heapsort und geben Sie als Zwischenschritte die Belegung von $a$ an, die vorliegt, bevor das Element $a[1]$ an seine endgültige Position $i$ getauscht wird.

**Aufgabe 2.9**

Überprüfen Sie, ob die folgenden Sortierverfahren stabil sind (d. h. die Reihenfolge von Elementen mit gleichem Sortierschlüssel wird während des Sortierverfahrens nicht vertauscht): Auswahlsort, Einfügesort, Shellsort, Mergesort, Radixsort, Bubblesort, Quicksort.

**Aufgabe 2.10**

Zeigen Sie: Lässt man als einzige Operation zwischen Schlüsseln Vergleichsoperationen zu, so benötigt man wenigstens $N-1$ Vergleiche im schlechtesten Fall um zwei sortierte Folgen

$$x_1 \leq x_2 \leq \ldots \leq x_{\frac{N}{2}} \quad \text{und} \quad y_1 \leq y_2 \leq \ldots \leq y_{\frac{N}{2}}$$

zu einer einzigen sortierten Folge $z_1 \leq z_2 \leq \ldots \leq z_N$ zu verschmelzen.

**Aufgabe 2.11**

Sortieren Sie die angegebene Zahlenfolge durch Fachverteilung. Geben Sie dabei die Belegung der einzelnen Fächer nach jeder Verteilphase an und jeweils die Folge, die nach einer Sammelphase entstanden ist.

$$1234, 2479, 7321, 4128, 5111, 4009, 6088, 9999, 7899, 6123,$$
$$3130, 4142, 7000, 0318, 8732, 3038, 5259, 4300, 8748, 6200$$

Wenn Sie die folgenden Zahlen zu sortieren hätten und dabei entweder Sortieren durch Fachverteilung oder ein Verfahren, das nur mit Schlüsselvergleichen arbeitet, verwenden könnten, welche Überlegungen würden Sie anstellen?

$$12345678, 43482809, 91929394, 91929390$$

**Aufgabe 2.12**

Berechnen Sie für die Schlüsselfolge $F_1$ die Vorsortierungsmaße $inv(F_1)$, $runs(F_1)$, $rem(F_1)$:

$$F_1 : 2,5,6,1,4,3,8,9,6$$

Berechnen Sie für die Schlüsselfolge $F_i$ $(i = 2,3,4)$ ebenfalls $inv(F_i)$, $runs(F_i)$, $rem(F_i)$, jeweils in Abhängigkeit von $N$ ($N$ gerade).

$$F_2 \quad : \quad 1, \frac{N}{2}+1, 2, \frac{N}{2}+2, \ldots, \frac{N}{2}$$
$$F_3 \quad : \quad N, \frac{N}{2}, N-1, \frac{N}{2}-1, \ldots, \frac{N}{2}+1, 1$$
$$F_4 \quad : \quad 1, N, 2, N-1, 3, \ldots \frac{N}{2}, \frac{N}{2}+1$$

**Aufgabe 2.13**

a) Geben Sie eine Folge $F$ von sieben Schlüsseln an, für die $inv(F) < runs(F)$ gilt.

b) Geben Sie eine Folge $F$ von sieben Schlüsseln an, für die $runs(F) < inv(F)$ gilt.

c) Geben Sie eine Folge von sieben Schlüsseln an, für die das natürliche 2-Wege-Mergesort möglichst wenige und das Verfahren A-sort möglichst viele Schlüsselvergleiche benötigt.

d) Geben Sie eine Folge $F$ von sieben Schlüsseln mit $runs(F) \leq 3$ an, für die das Verfahren A-sort möglichst viele Schlüsselvergleiche ausführt.

**Aufgabe 2.14**

Als weiteres Vorsortierungsmaß findet man in der Literatur $exc(F)$; das ist die kleinste Anzahl von Vertauschungen, die nötig sind um eine Folge $F$ in aufsteigende Ordnung zu bringen.

a) Geben Sie jeweils eine Folge $F$ mit $N$ Schlüsseln an, für die

   1. $exc(F)$ maximal wird;
   2. $exc(F) = \lfloor \frac{N}{2} \rfloor$ ist.

b) Welche Beziehung gilt zwischen $exc(F)$ und $inv(F)$ für alle Folgen $F$?

**Aufgabe 2.15**

Geben Sie allgemeine Bedingungen an, die für jedes Vorsortierungsmaß $m$ gelten sollten.

http://www.springer.com/978-3-662-55649-8

Algorithmen und Datenstrukturen
Ottmann, Th.; Widmayer, P.
2017, XXIV, 774 S. 364 Abb., Hardcover
ISBN: 978-3-662-55649-8