34 Rekursion

Bisher entwickelten wir Algorithmen auf der Basis von Programmstrukturen, die aus Folgen von Anweisungen, aus Verzweigungen und aus Schleifen bestanden. Schleifen wurden insbesondere dann verwandt, wenn Datenstrukturen mit einer variierenden Anzahl von Elementen iterativ zu verarbeiten waren. In diesem Kapitel wollen wir unser Inventar an Programmstrukturen um das Prinzip der Rekursion erweitern.

Rekursion

Definition 34-1: Rekursiver Algorithmus

Rekursiver Algorithmus

Ein Algorithmus ist rekursiv, wenn er Methoden (oder Funktionen) enthält, die sich selbst aufrufen.

Jede rekursive Lösung umfasst zwei grundlegende Teile:

Basisfall

- *den Basisfall, für den das Problem auf einfache Weise gelöst werden kann, und*

rekursive Definition

- *die rekursive Definition.*

Die rekursive Definition besteht aus drei Facetten:

1. *die Aufteilung des Problems in einfachere Teilprobleme,*
2. *die rekursive Anwendung auf alle Teilprobleme und*
3. *die Kombination der Teillösungen in eine Lösung für das Gesamtproblem.*

Bei der rekursiven Anwendung ist darauf zu achten, dass wir uns mit zunehmender Rekursionstiefe an den Basisfall annähern. Wir bezeichnen diese Eigenschaft als Konvergenz der Rekursion.

Konvergenz

□

Eine rekursive algorithmische Lösung bietet sich immer dann an, wenn man ein Problem in einfachere Teilprobleme aufspalten kann, die mit dem Originalproblem nahezu identisch sind und die man zuerst nach dem gleichen Verfahren löst.

Nehmen wir an, dass wir die Summe der ersten n natürlichen Zahlen berechnen wollen. Dies können wir mit unserem bisherigen Wissen iterativ mit Hilfe einer Schleife lösen.

```
int summeIterativ(int n) {
    int ergebnis = 0;
    for (int i = 1; i <= n; i++) {
        ergebnis += i;
    }
    return ergebnis;
}
```

Wir können es aber auch als rekursives Problem definieren, denn die Summe der ersten n Zahlen lässt sich berechnen, indem zunächst die Summe der ersten n-1

Zahlen berechnet wird und anschließend die n-te Zahl hinzuaddiert wird. Das Problem wird dadurch von n Zahlen auf n-1 Zahlen reduziert. Natürlich müssen wir auch einen Basisfall identifizieren. Dieser ist erreicht, wenn keine Zahl mehr übrig ist. Wir können das Problem folgendermaßen mathematisch beschreiben:

$$summe(n) = \begin{cases} 0 & \text{wenn } n == 0 \\ summe(n-1) + n & \text{sonst} \end{cases}$$

In Java können wir eine solche Lösung formulieren, indem wir die Methode selbst wieder aufrufen:

```
int summeRekursiv(int n) {
    // Basisfall: keine Zahl übrig
    if (n == 0) {
        return 0;
    }
    // sonst: rekursiver Aufruf
    return summeRekursiv(n - 1) + n;
}
```

Wird nun die Methode `summeRekursiv()` mit dem Wert 5 aufgerufen, so finden die folgenden Berechnungen statt:

```
summeRekursiv(5) =
summeRekursiv(5 - 1) + 5 =
(summeRekursiv(4 - 1) + 4) + 5 =
((summeRekursiv(3 - 1) + 3) + 4) + 5 =
(((summeRekursiv(2 - 1) + 2) + 3) + 4) + 5 =
((((summeRekursiv(1 - 1) + 1) + 2) + 3) + 4) + 5 =   // Basisfall
((((0 + 1) + 2) + 3) + 4) + 5 =
(((1 + 2) + 3) + 4) + 5 =
((3 + 3) + 4) + 5 =
(6 + 4) + 5 =
10 + 5 =
15
```

Selbsttestaufgabe 34-1:

Was passiert, wenn die Methode `summeRekursiv()` mit negativen Werten aufgerufen wird? Wie kann dieses Problem gelöst werden?

Fakultätsfunktion

Eine weiteres Beispiel ist die Fakultätsfunktion. Sie spielt bei vielen mathematischen Anwendungen eine wichtige Rolle. Sie wird typischerweise durch das Ausrufezeichen „!“ symbolisiert. Die Fakultätsberechnung ist für Eingabewerte $n \geq 0$ wie folgt definiert:

$$n! = \begin{cases} n * (n-1)! & \text{wenn } n > 1 \\ 1 & \text{wenn } n <= 1 \end{cases}$$

Selbsttestaufgabe 34-2:

Implementieren Sie eine Methode `fakultaetIterativ(int n)`*, die iterativ die Fakultät berechnet und eine Methode* `fakultaetRekursiv(int n)`*, die die Fakultät rekursiv berechnet. Die Methoden sollen dabei nur Werte* $n \geq 0$ *akzeptieren.*

Selbsttestaufgabe 34-3:

Begründen Sie, warum die Lösung für `fakultaetRekursiv()` *konvergiert.*

Selbsttestaufgabe 34-4:

Schreiben Sie die Methode `double power(int p, int n)`*, die für zwei ganze Zahlen* p *und* n *rekursiv den Wert* p^n *berechnet. Testen Sie Ihr Programm und vergleichen Sie es mit der Lösung von Selbsttestaufgabe 14.1-2, die eine iterative Variante aufzeigt.*

Die bisher betrachteten rekursiven Methoden hatten die Eigenschaft, dass sie nur einen rekursiven Aufruf pro Ablaufpfad beinhalten. Allerdings gibt es auch einige Probleme, die mehrere rekursive Aufrufe benötigen. Ein bekanntes Beispiel sind die Fibonacci-Zahlen. Die Fibonacci-Zahlen berechnen sich nach der folgenden Formel:

Fibonacci-Zahlen

$$fib(n) = \begin{cases} n & \text{wenn } 0 \leq n \leq 1 \\ fib(n-1) + fib(n-2) & \text{wenn } n > 1 \end{cases}$$

Selbsttestaufgabe 34-5:

Berechnen Sie alle Fibonacci-Zahlen bis $fib(8)$.

Die Implementierung einer passenden Methode ist nach dem gleichen Muster wie oben möglich. Negative Werte für `n` sind nicht zulässig; bei negativen Werten werfen wir eine `IllegalArgumentException`.

```
long fibRekursiv(int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Basisfall: n ist 0 oder 1
    if (n == 0 || n == 1) {
```

```
            return n;
        }
        // sonst: rekursiver Aufruf
        return fibRekursiv(n - 1) + fibRekursiv(n - 2);
    }
```

Selbsttestaufgabe 34-6:

Implementieren Sie eine iterative Lösung für die Berechnung der Fibonacci-Zahlen in der Methode `long fibIterativ(int n)`.

Selbsttestaufgabe 34-7:

In der Programmierung werden bisweilen Zufallszahlen benötigt. Es gibt Formeln zur Erzeugung sogenannter Pseudozufallszahlen, die eine Folge zufälliger Zahlen simulieren. Eine mögliche Formel zur Erzeugung solcher Zahlen ist die folgende:

Pseudozufallszahlen

$$f(n) = \begin{cases} n+1 & \textit{wenn } n < 3 \\ 1 + (((f(n-1) - f(n-2)) * f(n-3)) \textit{ mod } 100) & \textit{sonst} \end{cases}$$

Implementieren Sie eine Methode `long zufallszahl(int n)`*, die rekursiv* $f(n)$ *berechnet.*

Implementieren Sie außerdem eine Methode `void gebeZufallszahlenAus()`*, die die Pseudozufallszahlen* $f(5)$ *bis einschließlich* $f(30)$ *ausgibt.*

Bisher haben wir noch nicht weiter darüber nachgedacht, wie es funktionieren kann, dass eine Methode sich selbst aufruft. In Java wird bei einem Methodenaufruf ein Methodenrahmen erzeugt. In diesem Methodenrahmen sind alle aktuellen Werte der Parameter und lokalen Variablen gespeichert sowie ein Verweis auf die aufrufende Methode (siehe Abb. 34-1). Die Anweisungen einer Methode existieren nur einmal, sie sind nicht in jedem Methodenrahmen gespeichert.

Methodenrahmen

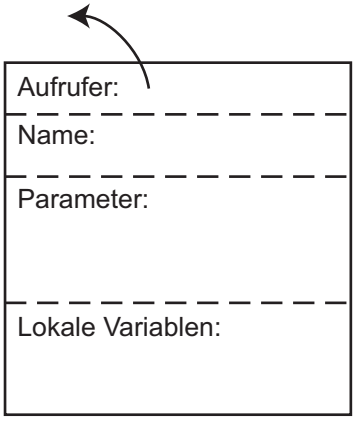

Abb. 34-1: Ein Methodenrahmen

Ruft nun eine Methode eine andere auf, so wird ein neuer Methodenrahmen erzeugt, und die Parameter und lokalen Variablen werden mit ihren Werten initialisiert. Dabei macht es keinen Unterschied, ob eine Methode eine andere oder eben sich selbst aufruft. Ist die aufgerufene Methode beendet, so wird dies dem Aufrufer – ggf. zusammen mit einem Rückgabewert – mitgeteilt.

Gegeben seien die beiden folgenden Methoden:

```
int a(int x) {
    int y = 2 * x;
    int z = 3;
    int w = b(y, z) + x;
    return w;
}

int b(int c, int d) {
    int e = c + 2 * d;
    return e;
}
```

Beim Aufruf der Methode `a()` mit dem Argument `3` wird ein Methodenrahmen für den Aufruf `a(3)` erzeugt (Abb. 34-2).

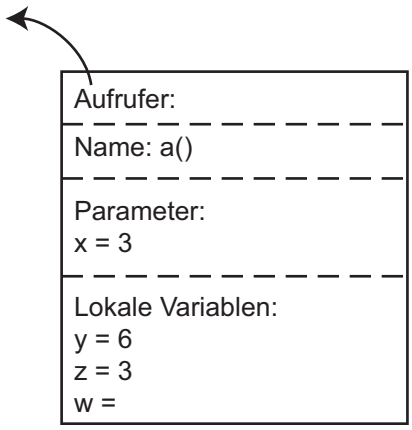

Abb. 34-2: Methodenrahmen für `a(3)`

Erreicht die Ausführung von `a(3)` den Aufruf von `b(6, 3)`, so wird auch dafür ein Methodenrahmen erzeugt (Abb. 34-3).

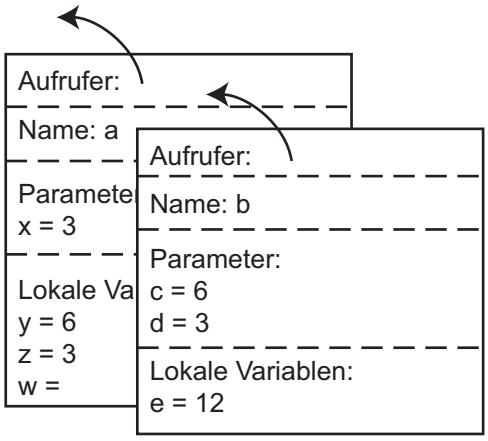

Abb. 34-3: Methodenrahmen für `a(3)` und `b(6, 3)`

Ist die Ausführung von `b(6,3)` beendet, kann der zugehörige Methodenrahmen wieder gelöscht werden und die Ausführung von `a(3)` wird an der entsprechenden Stelle fortgesetzt.

Bei einer rekursiven Methode passiert das gleiche, nur dass dort die Methodenrahmen alle von der gleichen Methode stammen. Sie werden jeweils mit eigenen Parametern und lokalen Variablen erzeugt. Abb. 34-4 veranschaulicht die Schritte bei der Ausführung von `summeRekursiv(4)`.

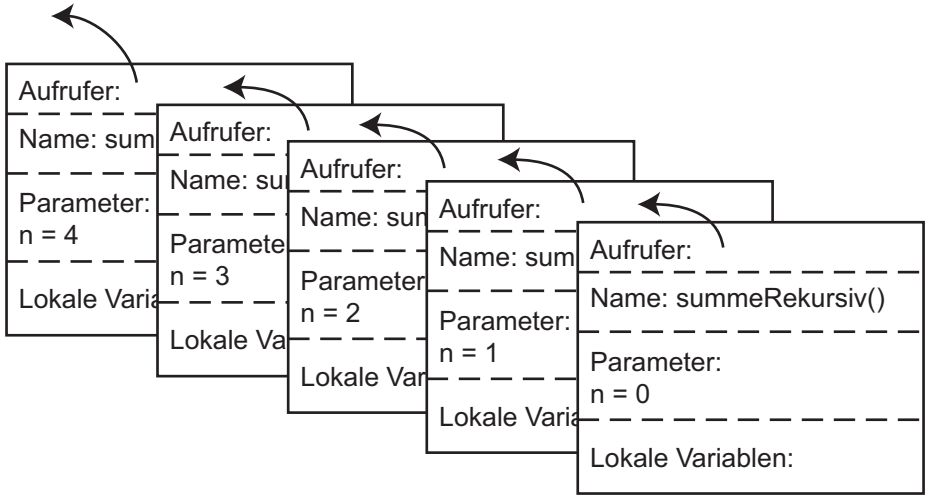

Abb. 34-4: Methodenstapel bei der Ausführung von `summeRekursiv(4)`

Wenn eine Methode eine andere aufruft, so entsteht ein sogenannter Methodenstapel (engl. stack). Vom Methodenstapel wird immer nur die oberste, also die zuletzt aufgerufene Methode ausgeführt. Erst wenn diese beendet ist und wieder vom Stapel entfernt wurde, kann die Methode darunter fortgesetzt werden. Die Datenstruktur des Stapels findet auch in anderen Bereichen Anwendung (siehe Kapitel 35).

Methodenstapel

Bemerkung: `StackOverflowError`

Wenn nicht mehr genug Speicherplatz für neue Methodenrahmen vorhanden ist, erzeugt die virtuelle Maschine in Java einen `StackOverflowError`*. Dieser Fall*

`StackOverflowError`

tritt auf, wenn eine Rekursion niemals den Basisfall erreicht oder den Basisfall erst nach zu vielen Schritten erreichen würde.

□

Rekursionen können nicht nur bei mathematischen Funktionen verwendet werden, sondern auch in vielen anderen Bereichen.

Palindrom

Ein Palindrom ist ein Wort, dass sowohl vorwärts als auch rückwärts gelesen das gleiche ist.

Selbsttestaufgabe 34-8:

Implementieren Sie die Methode `boolean istPalindromIterativ (String s)`*, die iterativ prüft ob es sich bei der Zeichenkette um ein Palindrom handelt.*

Der Begriff Palindrom lässt sich auch rekursiv definieren. Ein Wort aus einem oder keinen Zeichen ist immer ein Palindrom. Ein längeres Wort ist dann ein Palindrom, wenn der erste und letzte Buchstabe identisch sind und der Rest der Zeichenkette, also ohne den ersten und letzten Buchstaben, auch ein Palindrom ist.

Selbsttestaufgabe 34-9:

Implementieren Sie die Methode `boolean istPalindromRekursiv (String s)`*, die mit Hilfe der rekursiven Definition prüft, ob es sich bei der Zeichenkette um ein Palindrom handelt.*

Auch für das Suchen in und das Sortieren von Feldern gibt es einige rekursive Algorithmen. Bisher haben wir ein sortiertes Feld immer iterativ von einem Ende

lineare Suche

zum anderen durchsucht. Dieses Verfahren wird auch lineare Suche genannt. Wir können jedoch auch das mittlere Element eines sortierten Feldes betrachten und entscheiden, in welcher der beiden Hälften sich unser gesuchtes Element befindet. Anschließend halbieren wir die Hälfte wieder und treffen die gleiche Entscheidung. Wir gehen solange so vor, bis die zu durchsuchende Menge maximal noch 2 Elemente beinhaltet. Abb. 34-5 veranschaulicht dieses Verfahren.

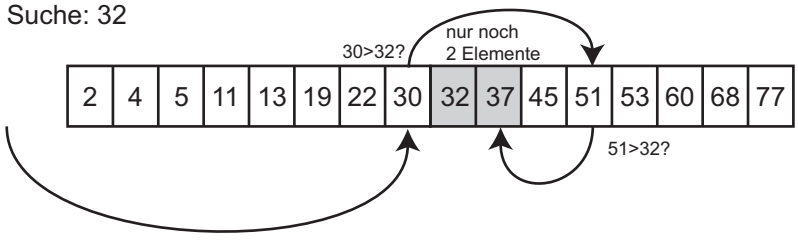

Abb. 34-5: Binäre Suche

Wir können die Methode `boolean istEnthalten(int wert, int[] feld, int start, int ende)`, die im Bereich `start` bis einschließlich `ende` im Feld nach dem Wert sucht, folgendermaßen rekursiv implementieren:

```
    // wir gehen von einem sortierten Feld aus
    boolean istEnthalten(int wert, int[] feld,
                                    int start, int ende) {
        // Basisfall: Bereich enthält maximal 2 Elemente
        if (ende - start <= 1) {
            return feld[start] == wert || feld[ende] == wert;
        }
        // sonst: rekursive Aufteilung
        // Mitte bestimmen
        int mitte = start + (ende - start) / 2;
        if (feld[mitte] == wert) {
            // wert gefunden
            return true;
        }
        if (feld[mitte] > wert) {
            // in linker Hälfte suchen
            return istEnthalten(wert, feld, start, mitte - 1);
        } else {
            // in rechter Hälfte suchen
            return istEnthalten(wert, feld, mitte + 1, ende);
        }
    }
```

Wir stellen fest, dass die Methode zwei Parameter hat, die für eine Überprüfung, ob ein Element in einem Feld enthalten ist, nicht wirklich notwendig sind. Wir können die Methode mit vier Parametern als `private` deklarieren und zusätzlich eine öffentliche Methode mit zwei Parametern anbieten, die die private Methode dann aufruft.

```
public class Binaersucher {

    public boolean istEnthalten(int wert, int[] feld) {
        return istEnthalten(wert, feld, 0, feld.length - 1);
    }

    private boolean istEnthalten(int wert, int[] feld,
                                    int start, int ende) {
        // ...
    }
}
```

Dieser Algorithmus wird als binäre Suche (engl. binary search) bezeichnet. Natürlich könnte auch schon bei größeren Restmengen auf ein anderes, zum Beispiel iteratives Suchverfahren umgeschaltet werden.

binäre Suche
binary search

Ein ähnliches Verfahren wenden wir auch häufig an, wenn wir zum Beispiel in einem Lexikon nach einem bestimmten Begriff suchen. Wir schlagen eine Seite auf, sehen nach, ob wir uns vor oder nach dem gesuchten Wort im Alphabet befinden, und wählen dann aus, in welchem Teil wir weitersuchen. Dabei lassen wir jedoch bei der Auswahl der nächsten Seite unser Wissen über die Position der Buchstaben im Alphabet mit einfließen, so dass wir nicht immer genau die Mitte des entsprechenden Teils auswählen. Dieses Wissen steht bei der binären Suche nicht zur Verfügung. Wird Wissen über die Verteilung bei der Suche berücksichtigt, so spricht man von einer Interpolationsuche.

Interpolationsuche

Selbsttestaufgabe 34-10:

Führen Sie auf dem folgenden Feld eine binäre und eine lineare Suche nach dem Element 38 aus. Wie viele Vergleiche benötigen Sie, bis Sie das Element gefunden haben?

```
int[] y = {3, 7, 14, 16, 18, 22, 27, 29, 30, 34, 38, 40, 50};
```

Selbsttestaufgabe 34-11:

Ergänzen Sie die Klasse `Binaersucher` *um eine Methode* `boolean istEnthalten(String s, String[] feld)`*, die mit Hilfe einer binären Suche prüft, ob die Zeichenkette in dem Feld enthalten ist. Sie können sich dabei Hilfsmethoden definieren. Hilfsmethoden sollten nach dem Geheimnisprinzip gekapselt sein.*

Sortieren von Feldern
Quicksort
Pivotelement

Für das Sortieren von Feldern gibt es beispielsweise den Algorithmus Quicksort. Bei diesem Verfahren wird zufällig ein Element des Feldes als sogenanntes Pivotelement ausgewählt. Anschließend werden die Elemente des Feldes aufgeteilt. In dem einen Teil befinden sich alle Elemente, die kleiner als das Pivotelement sind und im anderen alle, die größer als das Pivotelement sind. Anschließend werden beide Teile wiederum mit dem gleichen Verfahren aufgeteilt. Bei Teilen mit maximal zwei Elementen kann die Sortierung dann direkt vorgenommen werden. Anschließend ist das gesamte Feld sortiert. Abb. 34-6 veranschaulicht dieses Verfahren.

Um dieses Verfahren zu implementieren teilen wir den Algorithmus in zwei Teile. Das Aufteilen des Feldes implementieren wir in der Methode `aufteilen()` und das Sortieren in der Methode `quicksort()`

Dafür prüfen wir zunächst, ob das Feld weniger als 3 Elemente beinhaltet. Ist dies der Fall, so werden die beiden Elemente, wenn nötig, vertauscht. Bei einem größeren Feld wird das Feld mit Hilfe der Methode `aufteilen()` umsortiert. Anschließend werden die beiden Teile rekursiv mit dem gleichen Verfahren sortiert.

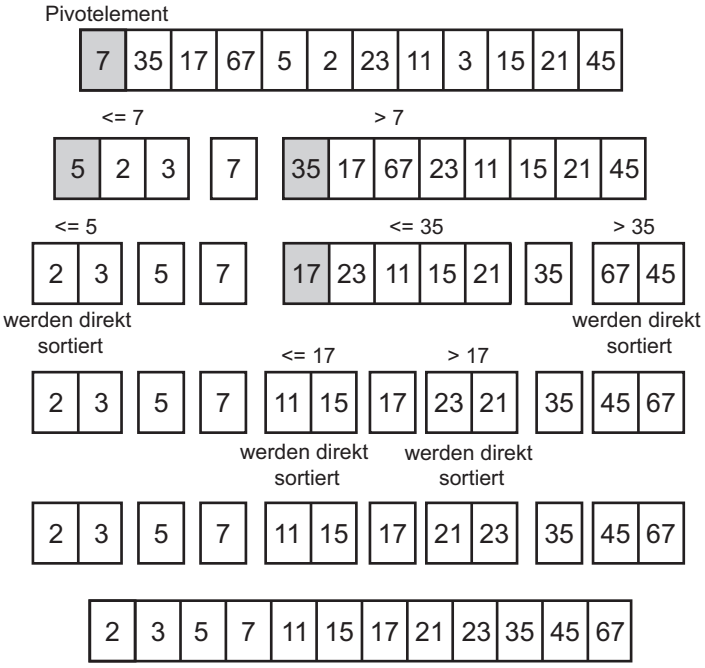

Abb. 34-6: Quicksort

Um zu wissen, in welchem Bereich sie sortieren soll, benötigt die Methode `quicksort()` zwei zusätzliche Parameter, ähnlich wie bei der binären Suche.

```
void quicksort(int[] feld, int start, int ende) {
    // Basisfall: leeres Feld
    if (ende < start) {
        return;
    }
    // Basisfall: maximal 2 Elemente,
    if (ende - start <= 1) {
        // wenn nötig die beiden Werte vertauschen
        if (feld[start] > feld[ende]) {
            int temp = feld[start];
            feld[start] = feld[ende];
            feld[ende] = temp;
        }
        return;
    }
    // Feld aufteilen
    int grenze = aufteilen(feld, start, ende);
    // linken Teil (ohne Pivot) Sortieren
    quicksort(feld, start, grenze - 1);
    // rechten Teil (ohne Pivot) Sortieren
    quicksort(feld, grenze + 1, ende);
}
```

Um das Feld entsprechend aufzuteilen, wählen wir das erste Element als Pivotelement. Anschließend beginnen wir, vom zweiten Element an aufsteigend, ein Element zu suchen, das größer als das Pivotelement ist. Ebenso beginnen wir, vom

letzten Element an absteigend, ein Element zu suchen, dass kleiner ist als das Pivotelement. Haben wir diese beiden Elemente gefunden, so vertauschen wir sie und suchen von den Positionen aus weiter. Das Vertauschen endet, sobald sich die Suchindizes von links und rechts treffen. Das Element an der Grenze wird anschließend mit dem Pivotelement vertauscht. Dieses Vorgehen ist in Abb. 34-7 dargestellt.

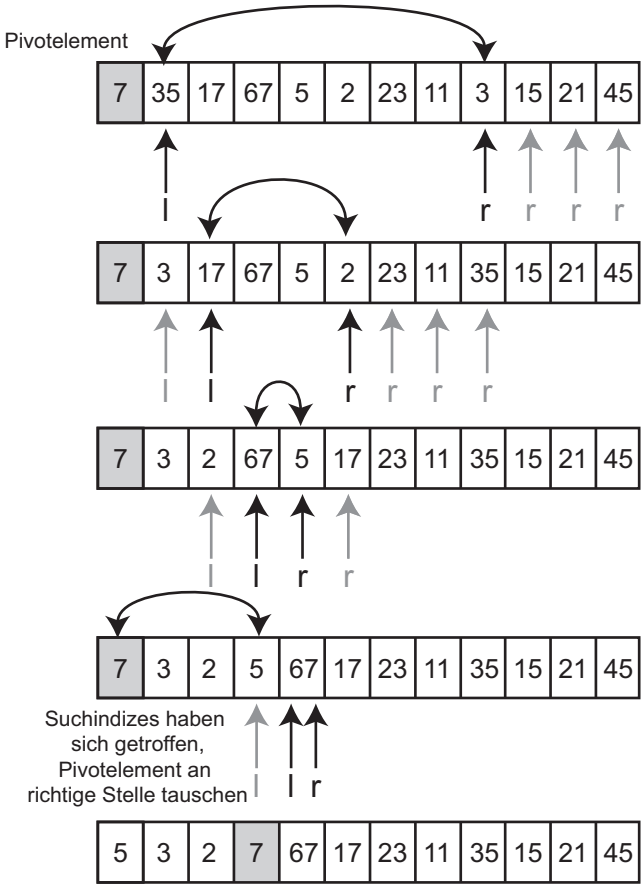

Abb. 34-7: Aufteilen eines Feldes an Hand des Pivotelements

```
// teilt die Elemente auf und liefert die
// Position des Pivotelements zurück
int aufteilen(int[] feld, int start, int ende) {
    // Index von links
    int l = start + 1;
    // Index von rechts
    int r = ende;
    // Pivotelement
    int pivot = feld[start];
    // Umsortierung
    while (l < r) {
        // erstes Element größer als Pivot finden
        while(feld[l] <= pivot && l < r) {
            l++;
        }
```

```
        // erstes Element kleiner als Pivot finden
        while(feld[r] > pivot && l < r) {
            r--;
        }
        // Elemente vertauschen
        int temp = feld[l];
        feld[l] = feld[r];
        feld[r] = temp;
    }
    // Indizes haben sich getroffen
    // prüfen ob Grenze korrekt
    if(feld[l] > pivot) {
        // Grenze anpassen
        l--;
    }
    // Grenze gefunden, Pivot entsprechend vertauschen
    feld[start] = feld[l];
    feld[l] = pivot;
    return l;
}
```

Selbsttestaufgabe 34-12:

Versuchen Sie, die einzelnen Schritte in der Implementierung mit Hilfe des folgenden Beispielaufrufs nachzuvollziehen.

```
int[] x = {12,2,6,1,8,34,10,7,20};
quicksort(feld, 0, feld.length - 1);
```

Sortieren durch Verschmelzen
merge sort

Ein weiteres rekursives Sortierverfahren ist das „Sortieren durch Verschmelzen“ (engl. merge sort). Bei diesem Verfahren wird im Gegensatz zu Quicksort nicht beim Aufteilen sortiert, sondern erst wieder beim Zusammenfügen. Das Feld wird in zwei möglichst gleich große Hälften aufgeteilt, die nach dem gleichen Verfahren sortiert werden. Die beiden sortierten Teilfelder werden nun zu einer sortierten Gesamtliste vereint, indem die beiden ersten Elemente verglichen und das kleinere aus seinem Teil entnommen und in das Zielfeld übernommen wird. Das wird solange fortgesetzt, bis beide Teilfelder leer sind. Abb. 34-9 veranschaulicht die Aufteilung und Verschmelzung der Felder.

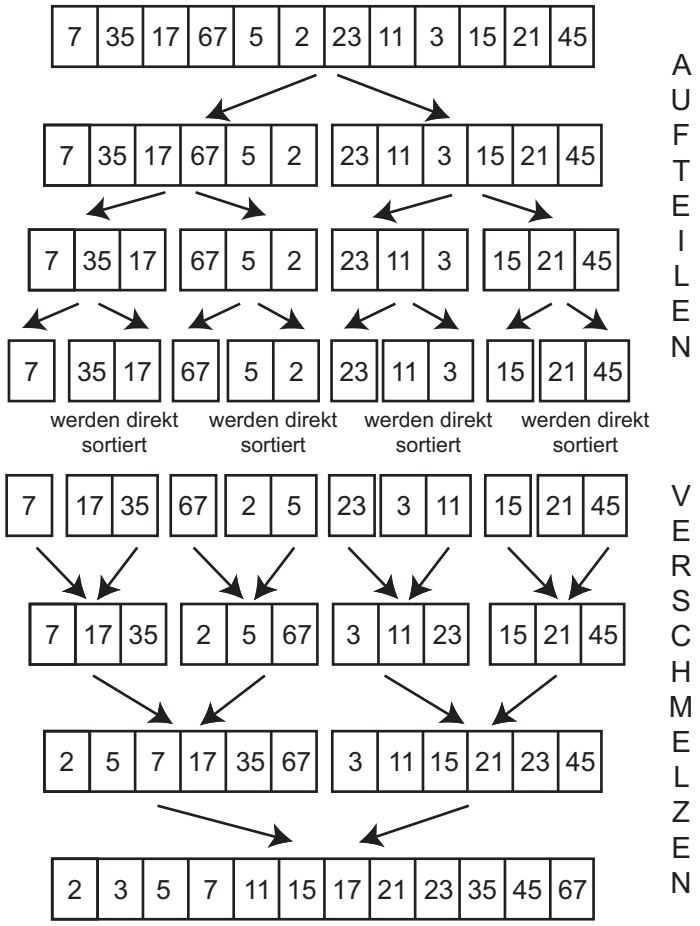

Abb. 34-9: Sortieren durch Verschmelzen (merge sort)

Das Verschmelzen der beiden Teilfelder $L = \langle l_1, l_2, \ldots, l_m \rangle$ und $R = \langle r_1, r_2, \ldots, r_n \rangle$ ist in Abb. 34-10 dargestellt und kann folgendermaßen definiert werden:

$$merge(L, R) = \begin{cases} L & \text{wenn } R \text{ leer ist} \\ R & \text{wenn } L \text{ leer ist} \\ l_1 + merge(\langle l_2, \ldots, l_m \rangle, R) & \text{wenn } l_1 \leq r_1 \\ r_1 + merge(L, \langle r_2, \ldots, r_n \rangle) & \text{sonst} \end{cases}$$

Das Zeichen „+“ soll hier bedeuten, dass das Element vorne in ein Feld eingefügt wird.

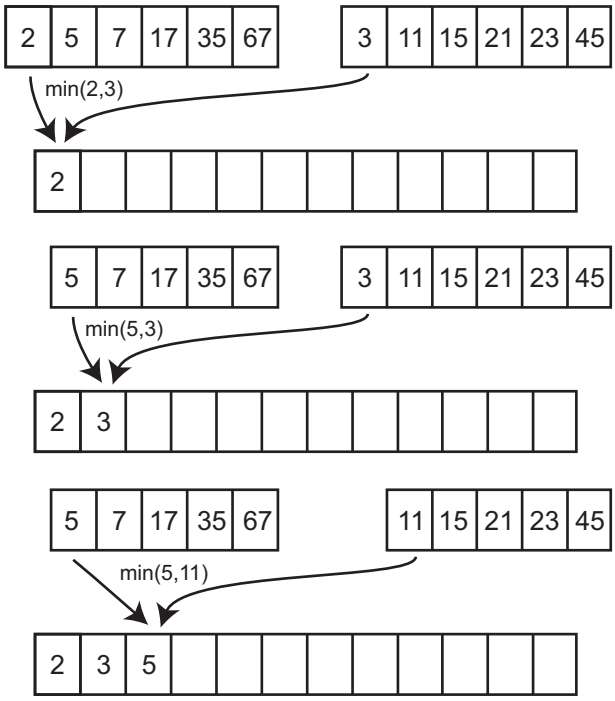

Abb. 34-10: Verschmelzen zweier sortierter Felder

Bei einer Implementierung würden wir feststellen, dass das Verschmelzen der Felder sehr mühsam ist, da wir immer wieder neue Felder erzeugen müssten, in die wir die Elemente hinein kopieren. Im folgenden Kapitel 35 werden wir Datenstrukturen kennen lernen, die die dafür notwendigen Operationen, wie zum Beispiel das Einfügen und Löschen von Elementen, das wir beim Verschmelzen benötigen, besser unterstützen.

Selbsttestaufgabe 34-13:

Versuchen Sie, das folgende Feld mit Hilfe des Algorithmus „Sortieren durch Verschmelzen“ von Hand zu sortieren.

```
int[] x = {12,2,6,1,8,34,10,7,20};
```