Prof. Dr. Bernd Krämer
Dr. Sebastian Küpper

Modul 63016

Einführung in die objektorientierte Programmierung für die Wirtschaftsinformatik

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Der Inhalt dieses Dokumentes darf ohne vorherige schriftliche Erlaubnis durch die FernUniversität in Hagen nicht (ganz oder teilweise) reproduziert, benutzt oder veröffentlicht werden. Das Copyright gilt für alle Formen der Speicherung und Reproduktion, in denen die vorliegenden Informationen eingeflossen sind, einschließlich und zwar ohne Begrenzung Magnetspeicher, Computerausdrucke und visuelle Anzeigen. Alle in diesem Dokument genannten Gebrauchsnamen, Handelsnamen und Warenbezeichnungen sind zumeist eingetragene Warenzeichen und urheberrechtlich geschützt. Warenzeichen, Patente oder Copyrights gelten gleich ohne ausdrückliche Nennung. In dieser Publikation enthaltene Informationen können ohne vorherige Ankündigung geändert werden.

Inhaltsverzeichnis

Kurseinheit 1

1 **Von der Aufgabenstellung zum Programm** 1
- 1.1 Motivation 1
- 1.2 Softwareentwicklung 2
- 1.3 EXKURS: Unified Modeling Language (UML) 4

2 **Anforderungsanalyse** 9
- 2.1 Fallstudie 9
- 2.2 Analyse der Anwendungswelt 10
- 2.3 Anwendungsfälle und Akteure 10
- 2.4 Beziehung zwischen Akteuren 13
- 2.5 Beziehung zwischen Anwendungsfällen 14
- 2.6 Anwendungsfallbeschreibungen 16
- 2.7 Datenlexikon 20
- 2.8 Pflichtenheft 21

3 **Einführung in die Objektorientierung** 23
- 3.1 Objekte und Klassen 23
- 3.2 Beziehungen 25
- 3.3 Kommunikation 28
- 3.4 Objektorientierte Analyse und Entwurf 29
- 3.5 UML-Klassendiagramm 31
- 3.6 UML-Objektdiagramm 33

4 **Einführung in die Algorithmik** 35
- 4.1 Algorithmen 35
- 4.2 Verhaltensbeschreibung und Kontrollstrukturen 40

5 **Programmiersprachen** 44
- 5.1 EXKURS: Entwicklungsgeschichte von Programmiersprachen 44
- 5.2 Eingabe, Verarbeitung, Ausgabe 46
- 5.3 Interaktive Programme 47

6 **Einführung in die Java-Programmierung** 48
- 6.1 Entwicklung und Eigenschaften der Programmiersprache Java 48
- 6.2 Erstellen, Übersetzen und Ausführen von Java-Programmen . 49

7 **Zusammenfassung** 53

Lösungshinweise **55**

Index **67**

Kurseinheit 2

8 **Praktischer Einstieg in die Java-Programmierung** 71
- 8.1 Ein erstes Programm 71
- 8.2 Klassen für die Praxis 73

8.3 Programmier- und Formatierhinweise 75
9 **Primitive Datentypen und Ausdrücke** 76
9.1 Ganze Zahlen 76
9.2 Gleitkommazahlen 79
9.3 Operatoren und Ausdrücke 81
9.4 Auswertung von Ausdrücken 83
9.5 Datentyp `boolean` 86
9.6 Datentyp `char` 87
10 **Variablen und Zuweisungen** 89
10.1 Variablen 89
10.2 Zuweisung 92
11 **Typanpassung** 98
11.1 Implizite Typanpassung 98
11.2 Explizite Typanpassung 103
12 **Anweisungen** 108
12.1 Blöcke 108
12.2 Kontrollstrukturen 110
13 **Bedingungsanweisungen** 111
13.1 Die einfache Fallunterscheidung 111
13.2 Die Mehrfach-Fallunterscheidung 116
14 **Wiederholungs- und Sprunganweisungen** 119
14.1 Bedingte Schleifen: `while` und `do` ... `while` 119
14.2 Die Zähl- oder `for`-Schleife 122
14.3 Strukturierte Sprunganweisungen: `break` und `continue` . 126
15 **Gültigkeitsbereich und Sichtbarkeit von lokalen Variablen** 131
16 **Zusammenfassung** 134

Lösungshinweise **137**

Index **153**

Kurseinheit 3

17 **Objekttypen** 157
17.1 Ein Objekt verwenden 157
17.2 Erzeugung und Lebensdauer von Objekten 161
17.3 Wertvariablen und Verweisvariablen 162
17.4 Zusammenspiel von Objekten 167
18 **Klassenelemente** 172
18.1 Klassenvereinbarung 172
18.2 Attributdeklaration 173
18.3 Methodendeklaration 174
18.4 Konstruktordeklaration 186
19 **Klassenvariablen und -methoden** 189
19.1 Klassenvariablen 189
19.2 Klassenmethoden 191

20 **Felder** ... 194
20.1 Felder einführen und belegen ... 194
20.2 Mehrdimensionale Felder ... 199
20.3 Erweiterte `for`-Schleife ... 203
21 **Zusammenfassung** ... 205

Lösungshinweise ... **207**

Index ... **229**

Kurseinheit 4

22 **Vererbung und Klassenhierarchien** ... 235
22.1 Vererbung ... 235
22.2 Vererbung in Java ... 236
22.3 Substitutionsprinzip ... 239
22.4 Überschreiben und Verdecken ... 244
22.5 Die Klasse Object ... 250
22.6 Konstruktoren und Erzeugung von Objekten einer Klasssenhierarchie ... 251
22.7 Polymorphie, dynamisches und statisches Binden ... 254
23 **Pakete** ... 260
23.1 Vereinbarung von Paketen ... 260
23.2 Klassennamen und Import ... 261
24 **Geheimnisprinzip und Zugriffskontrolle** ... 264
24.1 Geheimnisprinzip ... 264
24.2 Zugriffskontrolle bei Paketen und Klassen ... 264
24.3 Zugriffskontrolle bei Klassenelementen ... 266
25 **Abstrakte Einheiten** ... 277
25.1 Abstrakte Klassen und Methoden ... 277
25.2 Schnittstellen ... 283
26 **Zeichenketten** ... 289
26.1 Die Klasse `String` ... 289
26.2 Die Klasse `StringBuilder` ... 296
27 **Dokumentation** ... 299
28 **Zusammenfassung** ... 302

Lösungshinweise ... **305**

Index ... **319**

Kurseinheit 5

29 **Ausnahmen** ... 319
29.1 Ausnahmetypen ... 319
29.2 Ausnahmen erzeugen und werfen ... 321
29.3 Ausnahmen behandeln und weiterreichen ... 323

30 **Suchen und Sortieren** 329
30.1 Suchen in Feldern 329
30.2 Sortieren von Feldern 332
31 **Rekursion** 338
32 **Dynamische Programmierung** 353
32.1 Optimierungsprobleme und Anwendung der dynamischen Optimierung 356
32.2 Bedingungen für die Anwendbarkeit der dynamischen Programmierung 362
33 **Zusammenfassung** 368

Lösungshinweise **369**

Index **383**

Abbildungsverzeichnis

Abb. 29.1-1 Die Klasse Throwable und ihre wichtigsten Unterklassen 320
Abb. 28.3-1 Kontrollfluss in einer try-Anweisung 325
Abb. 33.1-1 Suchen einer Teilfolge in einem Feld 331
Abb. 33.2-1 Sortieren beim Einfügen (insertion sort) 334
Abb. 33.2-2 Bubblesort 335
Abb. 31-1 Ein Methodenrahmen 342
Abb. 31-2 Methodenrahmen für a(3) 342
Abb. 31-3 Methodenrahmen für a(3) und b(6, 3) 343
Abb. 31-4 Methodenstapel bei der Ausführung von summeRekursiv(4) 343
Abb. 31-5 Binäre Suche 345
Abb. 31-6 Quicksort 347
Abb. 31-7 Aufteilen eines Feldes an Hand des Pivotelements 348
Abb. 31-9 Sortieren durch Verschmelzen (merge sort) 350
Abb. 31-10 Verschmelzen zweier sortierter Felder 351
Abb. 32-1 : Darstellung der rekursiven Aufrufe zur Berechnung der fünften Fibonacci-Zahl 355
Abb. 32-2 : Darstellung der rekursiven Aufrufe zur Berechnung der fünften Fibonacci-Zahl in der Lösung mit dynamischer Programmierung 355
Abb. 32-3 : Darstellung der rekursiven Aufrufe zur Berechnung des Stabzerlegungsproblems für die Länge $n = 3$ 366
Abb. 31-8 Sortieren eines Beispielfeldes mit Quicksort 379
Abb. 31-11 Sortieren eines Beispielfeldes mit Sortieren durch Verschmelzen 380

Einführung

In den bisherigen Kurseinheiten haben wir gelernt, Klassen zu implementieren und zu dokumentieren. Dabei beschränkten wir uns auf relativ einfache Klassen und Methoden.

Häufig müssen Daten in großen Mengen verwaltet werden. Für solche Aufgaben verwendeten wir bisher Felder. Wir werden uns in dieser Kurseinheit damit beschäftigen wie wir in Feldern **suchen** und diese **sortieren** können.

Anschließend lernen wir das Prinzip der **Rekursion** kennen, das uns eine andere Herangehensweise für den Algorithmenentwurf bietet. Auch hierbei werden wir wieder auf Suchen und Sortieren von Feldern zurückkommen.

Ausgehend von der Beobachtung, dass bei rekursiven Lösungen Aufwand oft mehrfach anfällt, werden wir sehen, wie man mit **dynamischer Programmierung** den Rechenaufwand reduzieren kann. Dabei machen wir von der grundlegenden Methode der **Optimierung des Rechenaufwands auf Kosten des Speicherbedarfs** Gebrauch.

Lernziele

- Das Konzept der Ausnahmebehandlung erklären und Exceptions in Java einsetzen können.
- Das Prinzip der Rekursion kennen.
- Für verschiedene Probleme rekursive Lösungen formulieren und implementieren können.
- Das Konzept des Methodenrahmens und -stapels verstehen.
- Den Unterschied zwischen linearer und binärer Suche erläutern können und beiden Verfahren anwenden und implementieren können.
- Verschiedene Sortieralgorithmen kennen, anwenden und implementieren können.
- Die Elemente dynamischer Programmierung identifizieren und nachweisen können.
- Dynamische Programmierung zur effizienten Problemlösung einsetzen können.

29 Ausnahmen

Wir sind bei unseren bisherigen Beispielen auf einige Fälle gestoßen, wo nicht alle Werte als Argumente für Methoden zulässig sind. Wir wollten beispielsweise nicht zulassen, dass ein Rabatt negativ oder größer als 100% sein darf. Solche Situationen wurden entweder durch Ausgaben oder durch als ungültig interpretierte Rückgabewerte gekennzeichnet. Weitere typische Probleme sind der Zugriff auf ein Attribut oder eine Methode an einer Verweisvariable, der den Wert `null` liefert oder der Zugriff auf eine nicht existierende Datei.

Wir haben gesehen, dass in Programmen immer wieder problematische Situationen auftreten können. Java bietet Sprachkonstrukte, um mit Laufzeitfehlern konstruktiv umzugehen. Sie ermöglichen es,

- das Auftreten von Ausnahmesituationen zu überwachen,

Ausnahmesituation

- im Fall einer Ausnahmesituation, bei deren Auftreten der normale Verlauf der Programmabarbeitung unterbrochen wird, vom Laufzeitsystem ein Ausnahme-Objekt erzeugen zu lassen und

Ausnahme-Objekt

- die Kontrolle an spezielle Programmteile zu übergeben, die versuchen, den Laufzeitfehler aufzufangen (engl. catch) und zu behandeln.

Laufzeitfehler

Wir werden in diesem Kapitel die Erzeugung von Ausnahme-Objekten und die Behandlung von Ausnahmen kennen lernen. Als Beispiel verwenden wir die Methode `legeRabattFest()` der Klasse `Rechnung`, die wir dahingehend verbessern wollen, dass sie keinen negativen Rabatt und keinen Rabatt von über 100 Prozent akzeptiert.

29.1 Ausnahmetypen

Ausnahmeklassen
`Throwable`
`Exception`

In Java sind konkrete Ausnahmen [*JLS: § 11*] Objekte von bestimmen Ausnahmeklassen. Die allgemeinste Klasse ist die Klasse `Throwable`. Die beiden direkten Unterklassen von `Throwable` sind `Error` und `Exception` [*JLS: § 11.5*]. Objekte der Klasse `Exception` oder einer Unterklasse sind behandelbare Ausnahmen, wie zum Beispiel Probleme bei der Ein- und Ausgabe (`IOException`). Bei einer solchen Ausnahme soll das Programm nicht einfach abgebrochen werden, sondern es soll wenn möglich eine Lösung für das Problem gefunden werden. Lässt sich das Problem nicht beheben, so sollte eine entsprechende Mitteilung an den Benutzer erfolgen und das Programm ordentlich, d. h. mit einer `return`-Anweisung und nicht mit einer Ausnahme, beendet werden. Die Klasse `RuntimeException` ist eine direkte Unterklasse von `Exception`. Alle Objekte dieser Klasse oder einer ihrer Unterklassen, zum Beispiel `ArithmeticException` und `NullPointerException` spielen eine besondere Rolle, auf die wir später bei der Deklaration und Behandlung von

`RuntimeException`

Ausnahmen noch einmal zurückkommen werden. Die Klassenhierarchie ist in Abb. 28.1-1 angedeutet.

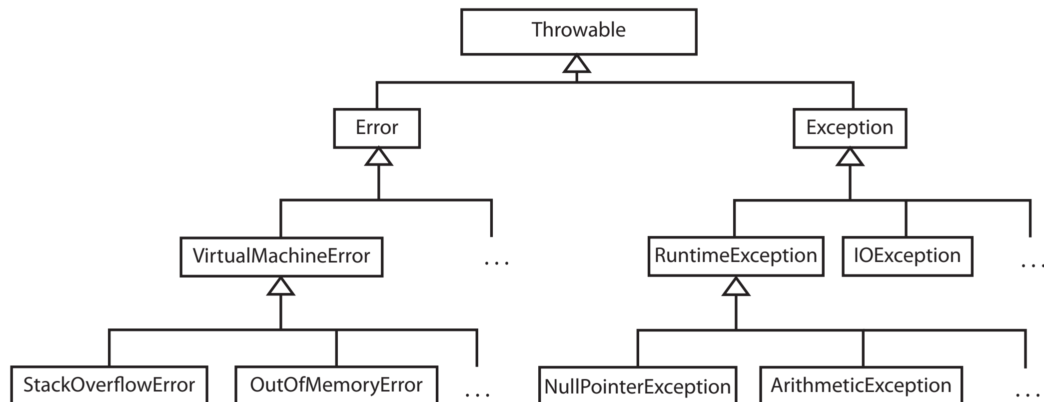

Abb. 28.1-1: Die Klasse `Throwable` und ihre wichtigsten Unterklassen

Ein Programmierer kann zusätzlich zu den bestehenden Ausnahmeklassen auch eigene Typen definieren. Eigene Ausnahmeklassen sind im Normalfall Unterklassen von `Exception`.

```
public class MeineAusnahme extends Exception {
}
```

getMessage()

Die Klasse `Exception` stellt schon einige hilfreiche Konstruktoren und Methoden zur Verfügung. Jede Ausnahme besitzt zur genaueren Beschreibung des Problems eine Nachricht (`message`). Diese kann, wenn gewünscht, mit Hilfe des Konstruktors der Oberklasse `Exception` festgelegt und mit Hilfe der Methode `getMessage()` erfragt werden. Wird die `toString()`-Methode aufgerufen, zum Beispiel durch eine Ausgabeanweisung, so werden der Name der Ausnahmeklasse und die Nachricht zurückgeliefert. Weitere Methoden der Klasse werden wir noch in den späteren Abschnitten kennen lernen.

Eine Ausnahmeklasse für einen ungültigen Rabatt könnte folgendermaßen aussehen:

```
public class UngueltigerRabattAusnahme extends Exception {
    public UngueltigerRabattAusnahme(double rabatt) {
        // setzt die Fehlermeldung
        super("Ein Rabatt von " + rabatt
               + " ist nicht zulaessig.");
    }
}
```

Nachdem wir nun wissen, welche Typen von Ausnahmen es gibt und wie wir eigene Typen definieren können, werden wir uns im nächsten Abschnitt damit beschäftigen, wie solche Ausnahmen in einem Java-Programm verwendet werden können.

29.2 Ausnahmen erzeugen und werfen

Wir wollen nun unsere Methode `legeRabattFest()` so ändern, dass sie keinen negativen Rabatt und keinen Rabatt von über 100 Prozent akzeptiert. Sie soll, falls sie mit einem unzulässigen Wert aufgerufen wird, eine Ausnahme erzeugen und diese werfen. Wird in einer Methode eine Ausnahme geworfen, so wird die Methode sofort mit einem Fehler beendet und dies dem Aufrufer mitgeteilt. Das Werfen einer Ausnahme geschieht mit Hilfe einer `throw`-Anweisung.

Definition 29.2-1: `throw`-Anweisung

Eine `throw`-Anweisung setzt sich aus dem Schlüsselwort `throw` und einer Referenz auf ein Ausnahmeobjekt, also ein Objekt vom Typ `Throwable` zusammen. [JLS: § 14.18]

`throw`-Anweisung

```
    throw Ausnahmeobjekt;
```

Häufig wird erst in der Anweisung mit Hilfe eines Konstruktoraufrufs ein neues Ausnahmeobjekt erzeugt. □

Kann eine Methode eine Ausnahme werfen, so muss sie dies auch in ihrem Methodenkopf angeben.

Definition 29.2-2: `throws` im Methodenkopf

Alle Ausnahmen, die eine Methode werfen kann, werden mit Hilfe des Schlüsselwortes `throws` nach der Parameterliste im Methodenkopf angegeben. [JLS: § 8.4.6, § 8.8.5]

`throws`

```
Modifikatoren Ergebnistyp Name(Parameterliste)
                                  throws Ausnahmetypenliste { }
```

Die `Ausnahmetypliste` besteht aus mindestens einem Ausnahmetyp. Weitere Ausnahmetypen können durch Kommata getrennt angegeben werden. □

Unsere Methode `legeRabattFest()` würde folglich nun folgendermaßen aussehen:

```
public class Rechnung {
    private double rabatt;
    // ...

    void legeRabattFest(final double neuerRabatt)
                           throws UngueltigerRabattAusnahme {
        if(neuerRabatt < 0 || neuerRabatt > 1) {
            throw new UngueltigerRabattAusnahme(neuerRabatt);
        }
```

```
        this.rabatt = neuerRabatt;
    }
}
```

Ausnahmen vom Typ `RuntimeException` müssen nicht explizit im Methodenkopf angegeben werden. Bei ihnen handelt es sich um nicht zu berücksichtigende Ausnahmen (engl. unchecked exceptions). [*JLS: § 11.2*]

nicht zu berücksichtigende Ausnahmen
`IllegalArgument-Exception`

Würden wir statt unserer selbst definierten Ausnahme die vordefinierte Ausnahme `IllegalArgumentException` nutzen, die eine Unterklasse von `RuntimeException` ist, so müssten wir folglich keinen `throws`-Teil im Methodenkopf angeben.

```
public class Rechnung {
    private double rabatt;
    // ...

    void legeRabattFest(final double neuerRabatt) {
        if(neuerRabatt < 0 || neuerRabatt > 1) {
            throw new IllegalArgumentException(
                    "Dieser Rabatt ist ungueltig: " + neuerRabatt);
        }
        this.rabatt = neuerRabatt;
    }
}
```

Bemerkung 29.2-1:

Kann eine Methode mehrere verschiedene Ausnahmen werfen, so können alle Typen einzeln oder nur entsprechende Oberklassen angegeben werden. Es ist besser, alle Ausnahmen explizit zu erwähnen, da so der Aufrufer auch geeignet auf jeden Typ reagieren kann.

Würden wir zwei Spezialisierungen von `UngueltigerRabattAusnahme` definieren, um zwischen negativen und zu hohen Rabatten unterscheiden zu können, so könnten im Methodenkopf entweder `UngueltigerRabattAusnahme` oder die beiden Spezialisierungen, mit Komma getrennt, aufgeführt werden.

```
public class NegativerRabattAusnahme
                               extends UngueltigerRabattAusnahme {
    public NegativerRabattAusnahme(double rabatt) {
        super(rabatt);
    }
}

public class ZuHoherRabattAusnahme
                               extends UngueltigerRabattAusnahme {
    public ZuHoherRabattAusnahme(double rabatt) {
        super(rabatt);
    }
}
```

```
public class Rechnung {
    private double rabatt;
    // ...

    void legeRabattFest(final double neuerRabatt) throws
                NegativerRabattAusnahme, ZuHoherRabattAusnahme {
        if(neuerRabatt < 0) {
            throw new NegativerRabattAusnahme(neuerRabatt);
        }
        if (neuerRabatt > 1) {
            throw new ZuHoherRabattAusnahme(neuerRabatt);
        }
        this.rabatt = neuerRabatt;
    }
}
```

□

Ausnahmen bieten somit die Möglichkeit, Methoden auf eine andere Art als mit einer `return`-Anweisung zu beenden und zusätzlich dem Aufrufer den Grund und weitere Informationen mitzuteilen.

Nachdem wir nun gelernt haben, Ausnahmen in Methoden zu werfen, werden wir uns im nächsten Abschnitt damit beschäftigen, wie verfahren werden muss, wenn eine Methode, die eine Ausnahme werfen kann, aufgerufen wird.

Selbsttestaufgabe 29.2-1:

Ergänzen Sie die Methode `legeMehrwertsteuerFest()` so, dass im Falle eines negativen Werts eine `IllegalArgumentException` mit einer passenden Fehlermeldung geworfen wird.

```
    void legeMehrwertsteuerFest(double neueMwSt) {
        this.mehrwertsteuer = neueMwSt;
    }
```

29.3 Ausnahmen behandeln und weiterreichen

Im Folgenden wollen wir uns nun damit beschäftigen, wie mit geworfenen Ausnahmen umgegangen wird. Nehmen wir an, wir erzeugen eine neue Rechnung und wollen anschließend einen Rabatt festlegen. Da die Methode eine Ausnahme werfen kann, müssen wir diese behandeln. Dies geschieht mit Hilfe der `try`-Anweisung.

Definition 29.3-1: `try`-Anweisung

`try`-Anweisung

Eine `try`-Anweisung besteht aus drei verschiedenen Bestandteilen. Die kritischen Anweisungen, wie zum Beispiele Aufrufe an Methoden, die Ausnahmen erzeugen können, werden in einen `try`-Block eingeschlossen. Nach diesem muss entweder mindestens ein `catch`-Abschnitt oder ein `finally`-Block folgen.

`catch`
`finally`

Mit Hilfe eines `catch`-Abschnitts können im `try`-Block aufgetretene Ausnahmen gefangen und behandelt werden. Im `catch`-Abschnitt wird angegeben, für welche Ausnahmetypen dieser Abschnitt zuständig ist. Sollen mehrere verschiedene Ausnahmetypen unterschiedlich behandelt werden, so können mehrere `catch`-Abschnitte folgen. Die Anweisungen im `finally`-Block werden immer am Ende der gesamten `try`-Anweisung ausgeführt. Hier werden in der Regel diverse Aufräumarbeiten erledigt. [JLS: § 14.20]

```
try {
    // Kritische Anweisungen
} catch (Ausnahmetyp1 name1) {
    // Diese Anweisungen werden ausgeführt, wenn
    // eine Ausnahme vom Typ Ausnahmetyp1 aufgetreten ist.
    // Die konkrete Ausnahme wird von der Variablen
    // name1 referenziert.
} catch (Ausnahmetyp2 name2) {
    // analog zum vorhergehenden Abschnitt
} finally {
    // Diese Anweisungen werden immer ausgeführt
}
```

Passen mehrere `catch`-Abschnitte zu der geworfenen Ausnahme, so wird immer nur der erste passende ausgeführt.

Abb. 29.3-1 veranschaulicht den Kontrollfluss in einer `try`-Anweisung. □

Wir führen in unserem Beispiel für die beiden Ausnahmen jeweils eine separate Behandlung durch. Einen `finally`-Block benötigen wir nicht:

```
public class AusnahmeBeispiel {
    public static void main(String[] args) {
        Rechnung r = new Rechnung();
        double rabatt = 0.2;
        try {
            r.legeRabattFest(rabatt);
        } catch (NegativerRabattAusnahme nra) {
            System.out.println(nra);
        } catch (ZuHoherRabattAusnahme zhra) {
            System.out.println(zhra);
        }
    }
}
```

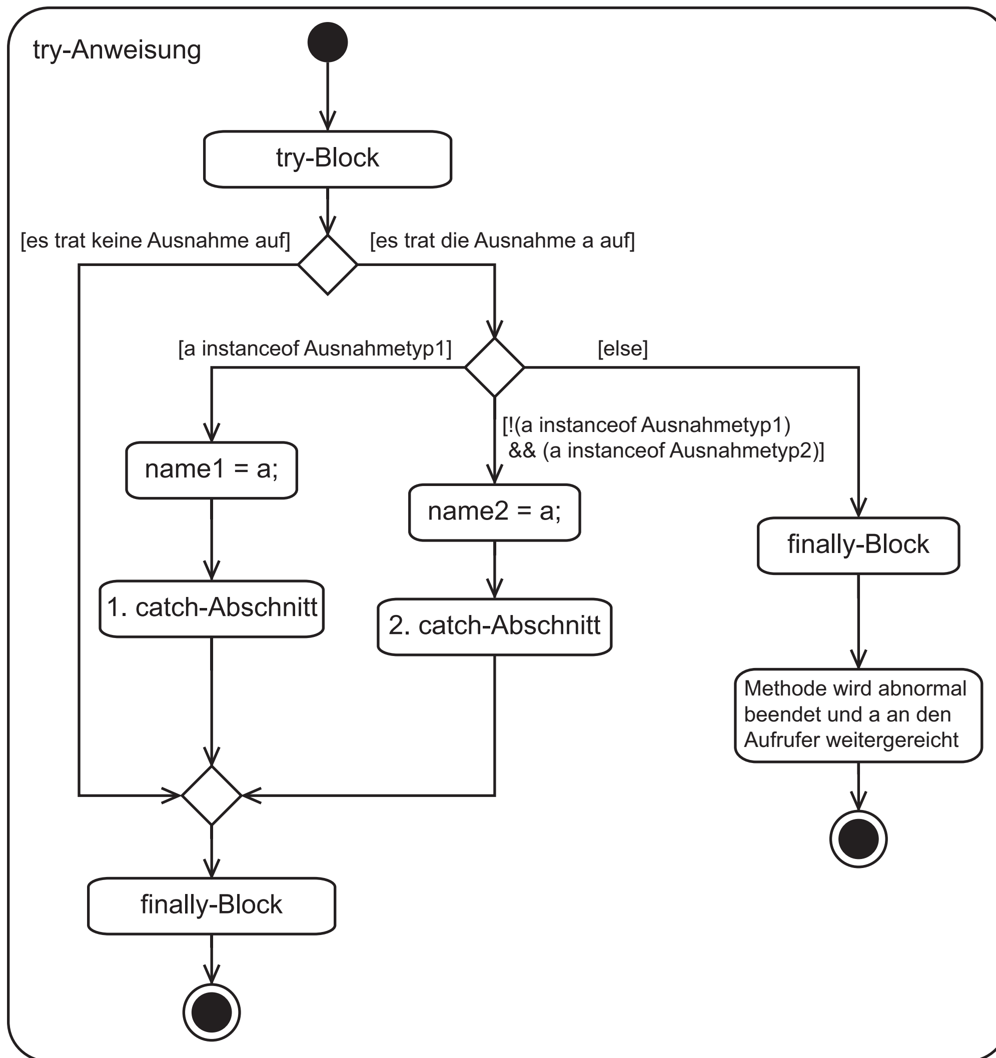

Abb. 29.3-1: Kontrollfluss in einer `try`-Anweisung

Bemerkung 29.3-1: Die Klasse `Error`

`Error`

Ein Objekt vom Typ `Error` ist im Normalfall ein sehr schwerwiegender Fehler, bei dem das Programm nur noch beendet werden kann. Beispiele dafür sind der `StackOverflowError` und der `OutOfMemoryError`, beides Unterklassen der Klasse `VirtualMachineError`. Exemplare der Klasse `Error` oder einer ihrer Unterklassen sollten somit nicht behandelt werden. Bei Ausnahmen vom Typ `Error` handelt es sich wie bei Ausnahmen vom Typ `RuntimeException` um nicht zu berücksichtigende Ausnahmen, so dass diese ebenfalls nicht mit `throws` im Methodenkopf angegeben werden müssen. [JLS: § 11.2.4] □

Selbsttestaufgabe 29.3-1:

Finden Sie heraus, welche Ausgaben erscheinen, wenn Sie für `rabatt` negative oder zu hohe Werte festlegen.

Selbsttestaufgabe 29.3-2:

Gegeben seien die folgenden Klassen:

```
public class MyException extends Exception {}

public class AnException extends MyException {}

public class AnotherException extends MyException {}

public class JustAnotherException extends MyException {}
```

Welcher der `catch`-Abschnitte wird ausgeführt, wenn der Methodenaufruf `foo()` die folgenden Ausnahmen wirft:

a. `AnException`

b. `JustAnotherException`

c. `java.io.IOException`

d. `AnotherException`

e. `java.lang.NullPointerException`

f. `MyException`

g. `java.lang.ArrayIndexOutOfBoundsException`

```
try {
    foo();
} catch (AnException e) {
    // 1
} catch (JustAnotherException e) {
    // 2
} catch (MyException e) {
    // 3
} catch (RuntimeException e) {
    // 4
} catch (Exception e) {
    // 5
}
```

Kann oder will eine aufrufende Methode die Ausnahme nicht behandeln, so kann sie diese auch an ihren Aufrufer weiterleiten. Dazu muss die Methode lediglich die Ausnahme ebenfalls in ihrem Methodenkopf mit Hilfe von `throws` angeben. Eine Ausnahme wird solange an die Aufrufer weitergereicht, bis eine Methode diese behandelt oder bis die `main()`-Methode erreicht wurde. In diese Falle bekommt der Benutzer die Ausnahme direkt angezeigt. Dass eine Ausnahme bis zum Benutzer gelangt, sollte in der Regel durch geeignete Behandlungen vermieden werden, da so dem Benutzer der Fehler besser erläutert werden kann.

```
public class Blumenladen {
    public void ausnahmeWeiterreichen() throws
                  NegativerRabattAusnahme, ZuHoherRabattAusnahme {
        Rechnung r = new Rechnung();
        double rabatt = 0.2;
        r.setRabatt(rabatt);
    }
}
```

Ausnahmen bieten noch weitere hilfreiche Informationen. So kann mit Hilfe der Methode `getStackTrace()` an einem Ausnahmeobjekt erfragt werden, in welcher Methode es geworfen und durch welche Methoden es weitergereicht wurde. Alternativ kann auch die Methode `printStackTrace()` genutzt werden, die den gesamten Methodenstapel auf der Fehlerausgabe ausgibt.

`getStackTrace()`

Ausnahmen bieten die Möglichkeit, Programme robuster zu machen und auf fehlerhafte Eingaben oder andere Probleme geeignet zu reagieren. Wir können unabhängig vom Ergebnistyp Fehler anzeigen, entsprechende Informationen mitgeben und auf die Ausnahmen geeignet reagieren.

Selbsttestaufgabe 29.3-3:

Finden Sie die Fehler in der Klasse `ExceptionTest`. Warum treten diese auf und wie können Sie diese beheben?

```
public class MyException extends Exception {}

public class AnException extends Exception {}

public class AnotherException extends RuntimeException {}

public class ExceptionTest {
    public void a(int x) {
        if (x < 0) {
            throw new MyException();
        }
        if (x == 0) {
            throw AnException();
        }
    }

    public void b(String y) {
        if (y == null) {
            throw new AnotherException();
        }
    }

    public void callA() {
        a(-2);
    }
```

```
    public void callB() {
        b(null);
    }
}
```

Selbsttestaufgabe 29.3-4:

Welche Ausgaben erzeugen die Aufrufe `z(-2)`, `z(0)` *und* `z(7)`*?*

```
class AnotherExceptionTest {
    public void z(int x) {
        System.out.println(a(x));
    }

    public int a(int x) {
        try {
            return b(x);
        } catch (RuntimeException e) {
            return -2;
        }
    }

    public int b(int x) {
        try {
            return c(x);
        } catch (NegativeValueException e) {
            return -1;
        }
    }

    public int c(int x) throws NegativeValueException,
                                 IllegalArgumentException {
        if (x < 0) {
            throw new NegativeValueException();
        } else if (x > 0) {
            return x * x;
        } else {
            throw new IllegalArgumentException(
                                  "0 is not a valid value");
        }
    }
}

class NegativeValueException extends Exception {
}
```

30 Suchen und Sortieren

Bisher haben wir uns primär mit einfachen Klassen und ihrem Entwurf beschäftigt und Techniken zur Qualitätssicherung kennen gelernt. Für reale Anwendungen werden jedoch meistens komplexere Daten und komplexeres Verhalten benötigt. Deshalb werden wir uns in diesem Kapitel mit gängigen Problemen und Algorithmen zum Thema Suchen und Sortieren beschäftigen. Diese Verfahren können auf verschiedenen Datenstrukturen angewendet werden; wir werden uns in diesem Kapitel zunächst auf Felder beschränken.

30.1 Suchen in Feldern

Suchen in Feldern

Suchen in strukturierten Datensammlungen ist eine häufig auftretende Programmieraufgabe, zu deren Lösung zahlreiche Algorithmen entwickelt wurden. Zunächst wollen wir eine Methode implementieren, die prüft, ob ein bestimmter Wert ist einem Feld enthalten ist. Dazu vergleichen wir jedes Element mit dem gesuchten Wert.

```
boolean istEnthalten(int wert, int[] feld) {
    for (int i : feld) {
        // prüfe ob gesuchter Wert
        if (wert == i) {
            // wenn Wert gefunden, Methode beenden
            return true;
        }
    }
    // Wert wurde nicht gefunden
    return false;
}
```

Wir können die Methode sofort verlassen, wenn wir den gesuchten Wert gefunden haben. Wird das Ende der Schleife erreicht, so wurde der Wert nicht gefunden.

Selbsttestaufgabe 30.1-1:

Implementieren Sie eine Methode `int bestimmeAnzahl(int wert, int[] feld)`*, die zählt, wie oft der übergebene Wert in dem Feld enthalten ist.*

Selbsttestaufgabe 30.1-2:

Implementieren Sie eine Methode `boolean istEnthalten(String wert, String[] feld)`*, die überprüft, ob die übergebene Zeichenkette im Feld enthalten ist. Was müssen Sie hierbei im Gegensatz zur Implementierung für Felder primitiver Typen beachten?*

Suche in sortierten Feldern

Bei der ersten Implementierung von `istEnthalten()` sind wir davon ausgegangen, dass die Elemente des Feldes unsortiert sind. Wenn wir wissen, dass das Feld aufsteigend sortiert ist, können wir die Methode beenden, sobald der aktuelle Wert größer als der gesuchte ist.

```
// Das Feld muss aufsteigend sortiert sein
boolean istEnthalten(int wert, int[] feld) {
    for (int i : feld) {
        // prüfe ob gesuchter Wert
        if (wert == i) {
            // wenn Wert gefunden, Methode beenden
            return true;
        }
        if (wert < i) {
            // der Wert kann im Rest des Feldes nicht
            // mehr vorkommen
            return false;
        }
    }
    // Wert wurde nicht gefunden
    return false;
}
```

Felder können nicht nur auf identische Elemente durchsucht werden, sondern auch auf Objekte, die bestimmte Eigenschaften aufweisen. So könnten beispielsweise alle Rechnungen einer bestimmten Kundin gesucht werden oder der Gesamtbetrag aller Rechnungen eines Kunden bestimmt werden.

Selbsttestaufgabe 30.1-3:

Gegeben seien die folgenden Klassen

```
public class Kunde {
    // ...
}

public class Rechnung {
    private Kunde rechnungsempfaenger;
    // ...

    public Kunde liefereRechnungsempfaenger() {
        return this.rechnungsempfaenger;
    }

    public int bestimmeBetragInCent() {
        // ...
    }

    // ...
}

public class Rechnungssammlung {
```

```
    private Rechnung[] rechnungen;

    // ...
}
```

Implementieren Sie eine Methode

`int bestimmeGesamtbetragAllerRechnungenVon(Kunde k)`

in der Klasse `Rechnungssammlung`*, die die Summe aller Rechnungsbeträge dieses Kunden in Cent bestimmt.*

Neben der Suche nach bestimmten Elementen kann auch das Maximum oder Minimum bestimmt werden.

Maximums- und Minimumssuche

Selbsttestaufgabe 30.1-4:

Implementieren Sie eine Methode `Rechnung findeTeuersteRechnung()` *in der Klasse* `Rechnungssammlung`*, die die Rechnung mit dem größten Betrag bestimmt.*

Alle Suchen können auch auf mehrdimensionalen Feldern durchgeführt werden.

Suchen auf mehrdimensionalen Feldern

Suchen von Teilfolgen

Bisher haben wir einzelne Elemente betrachtet, doch auch Teilfolgen können in einem Feld gefunden werden. So kann beispielsweise geprüft werden, ob eine bestimmte Zahlenfolge in einem Feld vorkommt. Man beginnt damit, den ersten Wert des Feldes mit dem ersten Wert der Zahlenfolge zu vergleichen. Stimmen die Werte überein, vergleicht man anschließend die zweiten Werte und so weiter. Stimmen die Werte nicht überein, so weiß man lediglich, dass die Zahlenfolge nicht beim ersten Element des Feldes beginnend gefunden werden kann. Man beginnt dann mit dem zweiten Element des Feldes und vergleicht es mit dem ersten der gesuchten Folge. Stimmen diese überein, so vergleicht man anschließend das dritte mit dem zweiten und so weiter. Abb. 30.1-1 veranschaulicht diese Schritte an einem Beispiel.

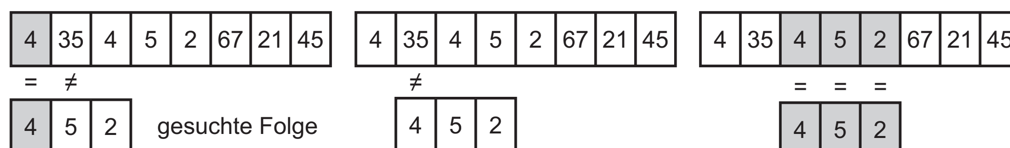

Abb. 30.1-1: Suchen einer Teilfolge in einem Feld

Wurde die gesamte Teilfolge gefunden, so kann die Methode beendet werden. Bei der Implementierung müssen wir aufpassen, dass wir nicht auf ungültige Feldelemente zugreifen.

```
boolean istTeilfolge(int[] feld, int[] gesuchteFolge) {
    // es muss nicht bis zum letzten Element gesucht werden,
    // da dort die Folge nicht mehr vollständig vorkommen kann
    for (int i = 0; i < feld.length
                        - gesuchteFolge.length + 1; i++) {
        for (int j = 0; j < gesuchteFolge.length; j++) {
            // Vergleiche entsprechende Einträge
            if (feld[i + j] != gesuchteFolge[j]) {
                // Zahl an Position j stimmt nicht überein
                // suche bei i+1 weiter
                break;
            } else if (j == gesuchteFolge.length - 1) {
                // gesamte Folge wurde gefunden
                return true;
            }
        }
    }
    // Teilfolge konnte nicht vollständig gefunden werden
    return false;
}
```

Selbsttestaufgabe 30.1-5:

Implementieren Sie eine Methode `int bestimmeAnfangdesWorts(char[] feld, String wort)`*, die die Position zurück liefert, ab der das Wort vollständig im Feld gefunden wurde. Wird das Wort nicht vollständig gefunden, so soll* `-1` *zurückgeliefert werden.*

Bemerkung 30.1-1: String-Matching

String-Matching

Das in Selbsttestaufgabe 30.1-5 beschriebene Problem wird auch als String-Matching bezeichnet. Zur Lösung dieses Problems existieren diverse Algorithmen.

□

Nachdem wir uns bisher mit der Suche in Feldern beschäftigt haben, werden wir uns im nächsten Abschnitt der Sortierung von Feldern widmen. Die beiden Probleme hängen insofern zusammen, als eine Suche auf sortierten Daten meistens effizienter lösbar ist.

30.2 Sortieren von Feldern

Sortieren von Feldern

Das Suchen in großen Datenbeständen wird erheblich beschleunigt, wenn die Datenelemente gemäß einer Ordnungsrelation sortiert wurden. Der Suchalgorithmus kann diese Eigenschaft nutzen, um ein bestimmtes Element in der sortierten Menge zu finden.

Beim Kartenspielen sortieren wir in der Regel unser Blatt nach dem Verfahren „Sortieren beim Einfügen“ (engl. insertion sort). Wir nehmen eine Karte auf und fügen sie an der richtigen Stelle des bereits sortierten Blatts ein. Dieses Verfahren können wir auch beim Sortieren der Elemente eines Feldes nachbilden. Wir betrachten das erste Element des Feldes als sortierte und den Rest der Elemente als unsortierte Menge. Nun vergleichen wir, mit dem zweiten Element beginnend, die nicht sortierten Elemente mit den Elementen mit kleinerem Index (die ja bereits sortiert sind) und fügen das betrachtete Element durch Vertauschen (engl. swap) von rechts nach links an der richtigen Stelle ein. Das Feld ist sortiert, wenn das Ende des Feldes erreicht ist.

Sortieren beim Einfügen
insertion sort

```
void sortiereAufsteigend(int[] feld) {
    // beginne beim zweiten Element und betrachte
    // die Liste bis zum Index i - 1 als sortiert
    for (int i = 1; i < feld.length; i++) {
        // gehe solange von i nach links
        // bis das Element an die richtige
        // Stelle gerutscht ist
        for (int j = i;  j > 0; j--) {
            if (feld[j - 1] > feld[j]) {
                // wenn linkes groesser,
                // vertausche die Elemente
                int temp = feld[j];
                feld[j] = feld[j - 1];
                feld[j - 1] = temp;
            } else {
                // das Element ist
                // an der richtigen Stelle
                break;
            }
        }
    }
}
```

Beachten Sie, dass das übergebene Feld direkt verändert wird und die Methode deshalb auch keinen Ergebnistyp benötigt.

Abb. 33.2-1 veranschaulicht das Verhalten des Sortieralgorithmus.

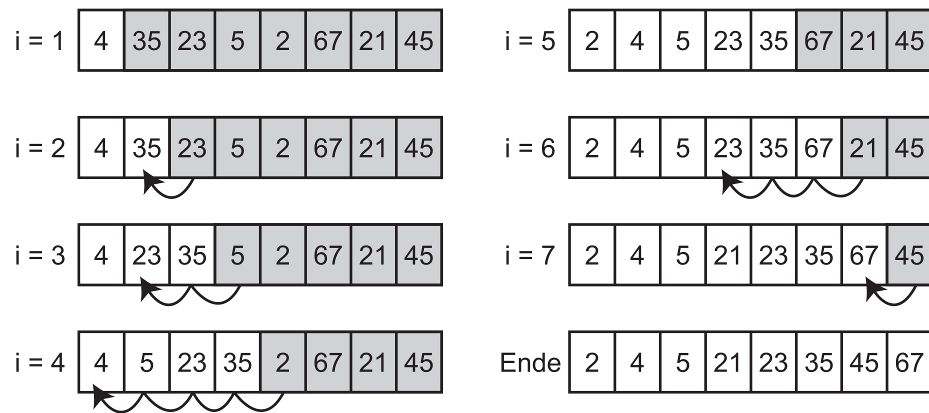

Abb. 33.2-1: Sortieren beim Einfügen (insertion sort)

Selbsttestaufgabe 30.2-1:

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus „Sortieren beim Einfügen“, um das folgende Feld zu sortieren?

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

Als Vergleich zählt die Auswertung der `if`-Bedingung.

Wir betrachten die drei Anweisungen im if-Block als eine Vertauschungsoperation.

Selbsttestaufgabe 30.2-2:

Entwerfen Sie analog eine Methode `void sortiereAbsteigend(String[] feld)`, die absteigend ein Feld von Zeichenketten sortiert.

Bubblesort

Ein weiterer klassischer Sortieralgorithmus ist Bubblesort. Bubblesort gleicht dem Sortieren durch Einfügen darin, dass benachbarte Elemente vertauscht werden, sofern sie nicht geordnet sind. Bubblesort vollzieht diese Vertauschungen aber in einer anderen Reihenfolge. Der Algorithmus ist der Beobachtung nachgebildet, dass sich verschieden große, in einer Flüssigkeit aufsteigende Blasen von selbst sortieren, weil die kleineren Blasen von den größeren beim Aufsteigen überholt werden. Der Algorithmus vergleicht zwei aufeinander folgende Feldelemente `feld[i]` und `feld[i + 1]` miteinander. Falls `feld[i] > feld[i + 1]` gilt, werden die Werte beiden Elemente miteinander vertauscht. Beginnend mit dem ersten Feldelement wird das Feld durchlaufen. Wenn nötig werden die beiden Elemente vertauscht. Dies wird solange von vorne wiederholt, bis keine Vertauschungen mehr notwendig sind, weil kein größerer Wert mehr vor einem kleineren vorkommt.

Das Verfahren ist in Abb. 33.2-2 veranschaulicht.

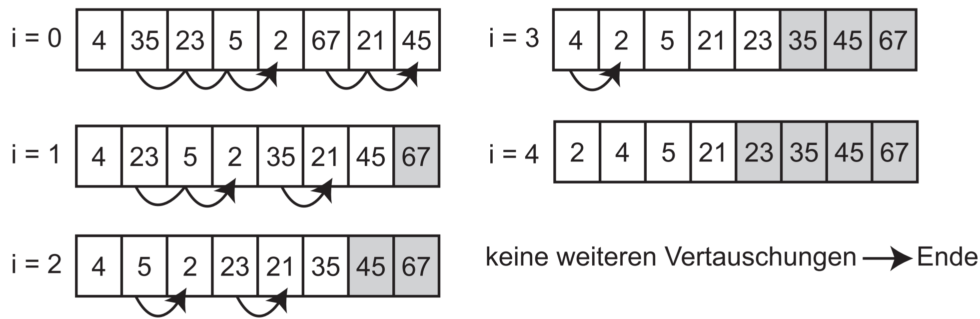

Abb. 33.2-2: Bubblesort

Wir stellen fest, dass nach dem ersten Felddurchlauf das größte Element an der richtigen Stelle steht. Im zweiten Durchlauf müssen wir also das letzte Element nicht mehr berücksichtigen. Nach dem zweiten Durchlauf steht dann das zweitgrößte Element an der richtigen Position, so dass im dritten Durchlauf die letzten beiden Elemente nicht mehr betrachtet werden müssen.

Sobald in einem Durchlauf keine Vertauschungen mehr durchgeführt wurden, ist das Feld vollständig sortiert.

```
void bubblesort(int[] feld) {
    // es werden  maximal feld.length - 1 Durchläufe benötigt
    for (int i = 0; i < feld.length - 1; i++) {
        // solange keine Vertauschungen vorgenommen werden
        // ist das Feld sortiert
        boolean sorted = true;
        // Durchlaufe das Feld, in jedem Durchlauf muss
        // ein Element weniger berücksichtigt werden
        for (int j = 0; j < feld.length - 1 - i; j++) {
            if (feld[j] > feld[j + 1]) {
                // wenn linkes größer
                // dann vertausche
                int temp = feld[j];
                feld[j] = feld[j + 1];
                feld[j + 1] = temp;
                // Feld ist nicht sortiert
                sorted = false;
            }
        }
        if (sorted) {
            // keine Vertauschungen, Feld ist
            // folglich vollständig sortiert
            break;
        }
    }
}
```

Selbsttestaufgabe 30.2-3:

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus Bubblesort, um das folgende Feld zu sortieren?

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

Als Vergleich zählt die Auswertung der `if`-Bedingung.

Wir betrachten die drei Anweisungen im inneren if-Block als eine Vertauschungsoperation.

Selbsttestaufgabe 30.2-4:

Entwerfen Sie mit Hilfe des Bubblesort-Algorithmus eine Methode `void sortiereAufsteigend(Rechnung[] rechnungen)`, die die Rechnungen aufsteigend nach ihren Beträgen sortiert.

Selbsttestaufgabe 30.2-5:

Sortieren durch Auswählen
selection sort

Der Algorithmus „Sortieren durch Auswählen“ (engl. selection sort) nimmt am Anfang das komplette Feld als unsortiert an, sucht sich das größte Element unter den unsortierten Elementen und vertauscht es, wenn nötig, anschließend mit dem letzten Element des unsortierten Bereichs.

Im nächsten Schritt gehört das letzte Element nun zum sortierten Bereich. Es wird wiederum das größte Element des unsortierten Bereichs mit dem letzten Element vertauscht, so dass es anschließend zum sortierten Bereich gehört. Der Algorithmus terminiert, wenn der unsortierte Bereich aus einem Element besteht.

Wenden Sie den Algorithmus auf das folgende Feld an und zählen Sie die dabei nötigen Vergleichs- und Vertauschungsoperationen .

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

Selbsttestaufgabe 30.2-6:

Implementieren Sie die Methode `void sortiere(double[] feld)` so, dass sie das übergebene Feld mit Hilfe des Algorithmus „Sortieren durch Auswählen“ sortiert.

Selbsttestaufgabe 30.2-7:

Vergleichen Sie die bei den verschiedenen Sortieralgorithmen notwendigen Vergleichs- und Vertauschungsoperationen für die folgenden Felder.

```
int[] beispiel1 = {4, 35, 23, 5, 2, 67, 45, 21};
int[] beispiel2 = {2, 4, 5, 21, 23, 35, 45, 67};
int[] beispiel3 = {67, 45, 35, 23, 21, 5, 4, 2};
```

Was fällt Ihnen dabei auf?

Weiterführende Literatur

[Sedgewick03] Robert Sedgewick:
Algorithms in Java
Addison Wesley 2003

Der Autor hat schon zahlreiche Bücher zum gleichen Thema in englischer Sprache verfasst. Das Werk ist didaktisch gut aufgebaut und enthält zahlreiche Illustrationen und Programmcode in Java.

[Cormen09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein:
Introduction to Algorithms, Third Edition
MIT Press 2009

31 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 31-1: 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 hinzu addiert 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 31-1:

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:

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

Selbsttestaufgabe 31-2:

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 31-3:

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

Selbsttestaufgabe 31-4:

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 31-5:

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 31-6:

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

Selbsttestaufgabe 31-7:

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

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

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, sowie ein Verweis auf die aufrufende Methode (siehe Abb. 31-1) gespeichert. Die Anweisungen einer Methode existieren nur einmal, sie sind nicht in jedem Methodenrahmen gespeichert.

Methodenrahmen

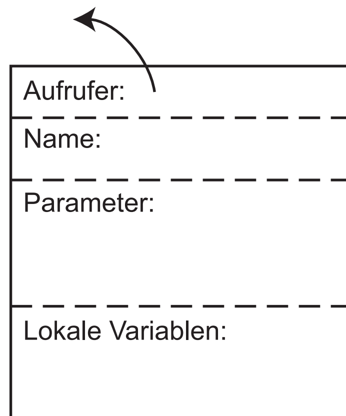

Abb. 31-1: Ein Methodenrahmen

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. 31-2).

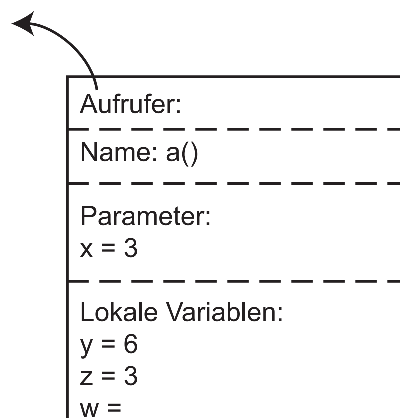

Abb. 31-2: Methodenrahmen für `a(3)`

Erreicht die Ausführung von `a(3)` den Aufruf von `b(6, 3)`, so wird auch dafür ein Methodenrahmen erzeugt (Abb. 31-3).

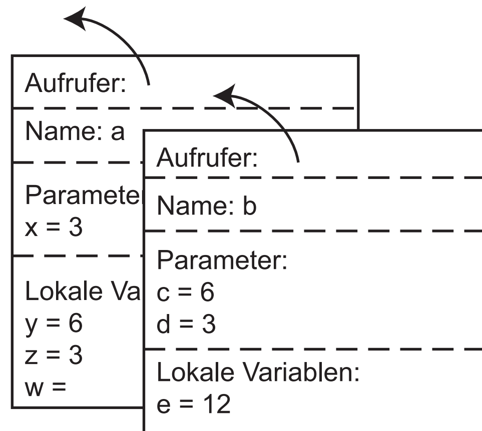

Abb. 31-3: Methodenrahmen für `a(3)` und `b(6, 3)`

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. 31-4 veranschaulicht die Schritte bei der Ausführung von `summeRekursiv(4)`.

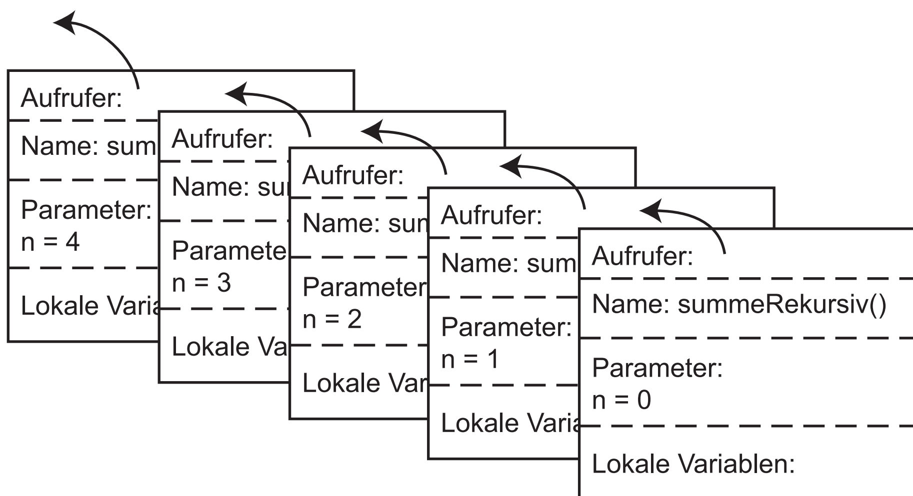

Abb. 31-4: Methodenstapel bei der Ausführung von `summeRekursiv(4)`

Methodenstapel

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 34 in Kurseinheit 6).

Bemerkung: `StackOverflowError`

StackOverflowError

Wenn nicht mehr genug Speicherplatz für neue Methodenrahmen vorhanden ist, erzeugt die virtuelle Maschine in Java einen `StackOverflowError`. Dieser Fall 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 31-8:

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 31-9:

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.

lineare Suche

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 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. 31-5 veranschaulicht dieses Verfahren.

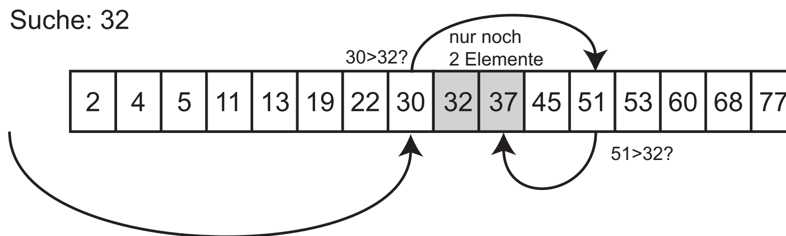

Abb. 31-5: Binäre Suche

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) {
        // ...
```

```
    }
}
```

binäre Suche
binary search

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.

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 weiter suchen. 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 Interpolationssuche.

Interpolationssuche

Selbsttestaufgabe 31-10:

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 31-11:

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. 31-6 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()`.

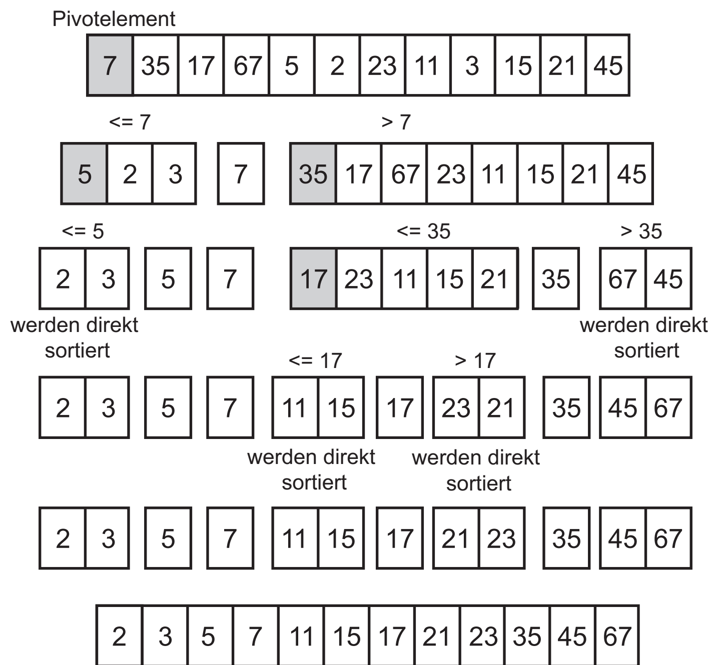

Abb. 31-6: Quicksort

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.

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, das 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. 31-7 dargestellt.

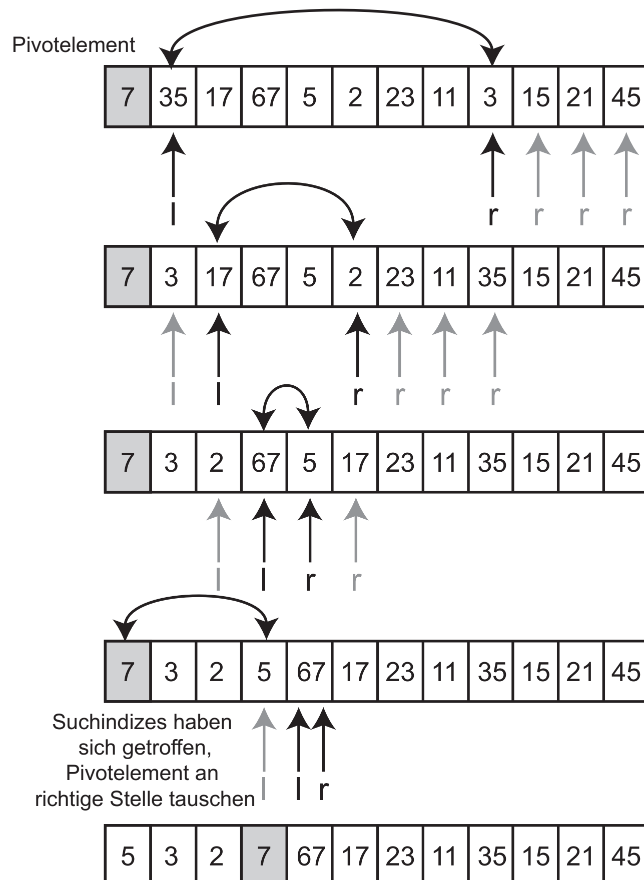

Abb. 31-7: Aufteilen eines Feldes an Hand des Pivotelements

```
// 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 31-12:

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. 31-9 veranschaulicht die Aufteilung und Verschmelzung der Felder.

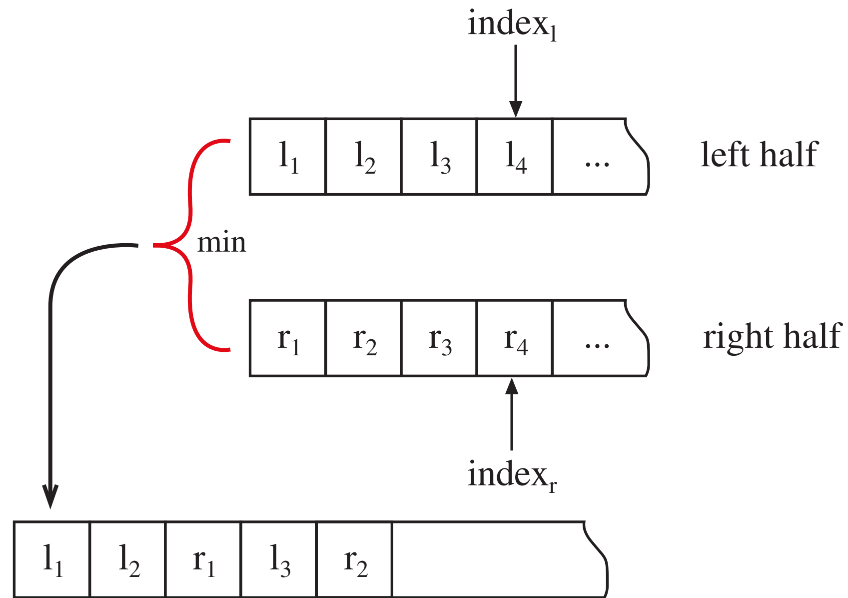

Abb. 31-9: Sortieren durch Verschmelzen (merge sort)

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. 31-10 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.

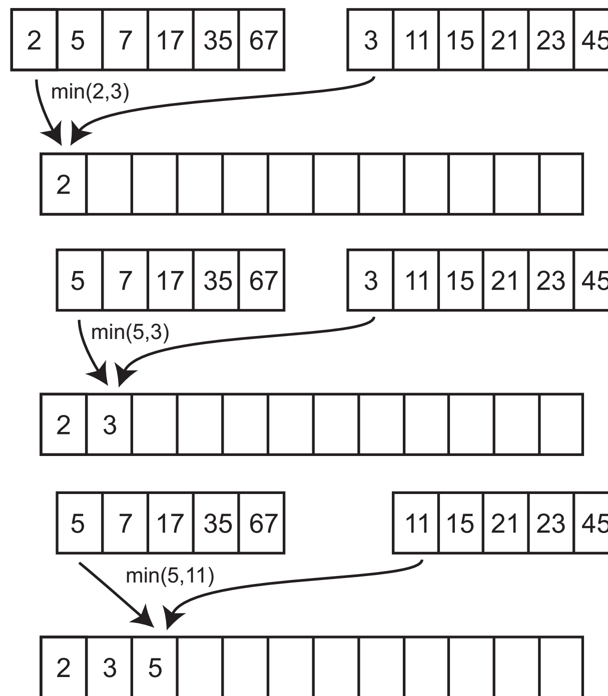

Abb. 31-10: Verschmelzen zweier sortierter Felder

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 Kapitel 34 der Kurseinheit 6 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 31-13:

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};
```

Quicksort und Sortieren durch Verschmelzen sind sicherlich auf den ersten Blick weniger intuitiv als die zuvor besprochenen Sortierverfahren wie Sortieren beim Einfügen oder Bubblesort. Dass diese Verfahren dennoch in der Praxis Anwendung finden, kann man anhand der folgenden Überlegung nachvollziehen:

Wie lange benötigt Sortieren beim Einfügen um eine Liste der Länge n zu sortieren? Das hängt natürlich stark von der Eingabe ab. Der ungünstigste Fall für dieses Verfahren ist gegeben, wenn man ein Feld von n Zahlen sortieren möchte, die absteigend sortiert sind. In diesem Fall müssen wir bei jeder Einfügeoperation die komplette Ergebnisliste durchlaufen. Wir benötigen also

$$0+1+2+3+...+n-1 = \sum_{i=0}^{n-1} i = \sum_{i=1}^{n} i - n = (n+1) \cdot \frac{n}{2} - n = \frac{n^2}{2} + \frac{n}{2} - n = \frac{n^2}{2} - \frac{n}{2}$$

Vergleichsschritte. Der dominante Term, der hier das Laufzeitverhalten charakteristisch beschreibt [1] ist n^2. Die selbe Analyse und das selbe Beispiel zeigen das gleiche Laufzeitverhalten im ungünstigsten Fall für Bubblesort. Man kann außerdem zeigen, dass auch im durchschnittlichen Fall die Zahl der Vergleichsoperationen in der Größenordnung von n^2 liegt. Im günstigsten Fall, der vollständig sortierten Liste, hingegen werden nur n Vergleichsoperationen benötigt.

Sortieren durch Verschmelzen (engl. Mergesort) hingegen benötigt, unabhängig von der Eingabe, in jeder Rekursionsebene n Vergleichsoperationen, um die Verschmelzung durchzuführen. Da die Unterteilung wahllos in der Hälfte des Feldes erfolgt, ist hierzu gar keine Vergleichsoperation notwendig. Insgesamt gibt es $\lceil \log_2(n) \rceil$ Ebenen, die Zahl der Vergleichsoperationen von Sortieren durch Verschmelzen ist also (im günstigsten, ungünstigsten und mittleren) Fall stets $\log_2 n \cdot n$. Im Durchschnitt und im schlimmsten Fall ist Sortieren durch Verschmelzen also (für große zu sortierende Felder) signifikant schneller als die einfacheren Verfahren. Man beachte aber, dass Sortieren durch Verschmelzen keinen Vorteil daraus ziehen kann, wenn eine Liste bereits (beinahe) sortiert ist.

Quicksort hingegen liegt in der gleichen Größenordnung wie Sortieren durch Verschmelzen was die Zahl der Vergleichsoperationen im mittleren Fall und im besten Fall anbelangt; im schlechtesten Fall – der durch eine vollständig sortierte Liste erreicht werden kann – kann Quicksort allerdings mit Sortieren durch Verschmelzen nicht mithalten und benötigt in der Größenordnung von n^2 Vergleichsoperationen. Dadurch, dass der Verwaltungsaufwand bei Quicksort wesentlich geringer ist als bei Sortieren durch Verschmelzen, wird in der Praxis dennoch oft Quicksort statt Sortieren durch Verschmelzen verwendet. Insgesamt haben die verschiedenen Sortierverfahren verschiedene Vorteile, so dass es sich lohnt, Bubblesort, Quicksort und Sortieren durch Verschmelzen als Implementation bereitzuhalten:

- Für nahezu sortierte oder kurze Feldern lohnt sich Bubblesort, da es den geringsten Verwaltungsaufwand hat und nahezu sortierte Felder schnell sortiert.
- Bei größeren Feldern unbekannter Sortierung ist Quicksort der Algorithmus der Wahl, da es eine niedrige durchschnittliche Zahl an Vergleichsoperationen mit einem moderaten Verwaltungsaufwand kombiniert.
- Bei sehr großen Feldern schließlich fallen die Nachteile des höheren Verwaltungsaufwands von Sortieren durch Verschmelzen nicht mehr ins Gewicht und die Zahl der Vergleichsoperationen dominiert das Laufzeitverhalten. Dann lohnt es sich, zur Vermeidung der ungünstigen Fälle bei Quicksort, Sortieren durch Verschmelzen zu verwenden.

---

1 Der Grund hierfür ist, dass mit steigendem n Der Teilterm n^2 den Teilterm n dominiert. Konstante Vorfaktoren wie hier $\frac{1}{2}$ werden bei Laufzeitanalysen zwecks Ausbildung sprechender Klassen zudem üblicherweise ignoriert. Die so genannte Landau-Notation, die es erlaubt die hier errechnete Zahl an Vergleichsoperationen als $O(n^2)$ zu schreiben, werden Sie im Kurs „Einführung in die technischen und theoretischen Grundlagen der Informatik“ kennen lernen.

32 Dynamische Programmierung

Bei der Entwicklung rekursiver Lösungen haben wir bislang nicht berücksichtigt, ob wir gegebenenfalls durch die rekursiven Abstieg Teillösungen mehr als ein Mal berechnen. Allerdings ist das keine unerhebliche Fragestellung, denn der Rechenaufwand, der durch wiederholtes Berechnen der gleichen Teillösungen anfällt, kann signifikant sein.

Betrachten wir beispielsweise unser Programm zur Berechnung der Fibonacci-Zahlen, am Beispiel der Eingabe 5. Um die fünfte Fibonacci-Zahl zu berechnen, benötigen wir die vierte und die dritte Fibonacci-Zahl. Um die vierte Fibonacci-Zahl zu berechnen, benötigen wir wiederum die dritte und die zweite Fibonacci-Zahl. Die folgende Grafik stellt den entsprechenden Aufrufbaum dar, wir schreiben zur einfacheren Darstellung $f(_)$ für $fib(_)$:

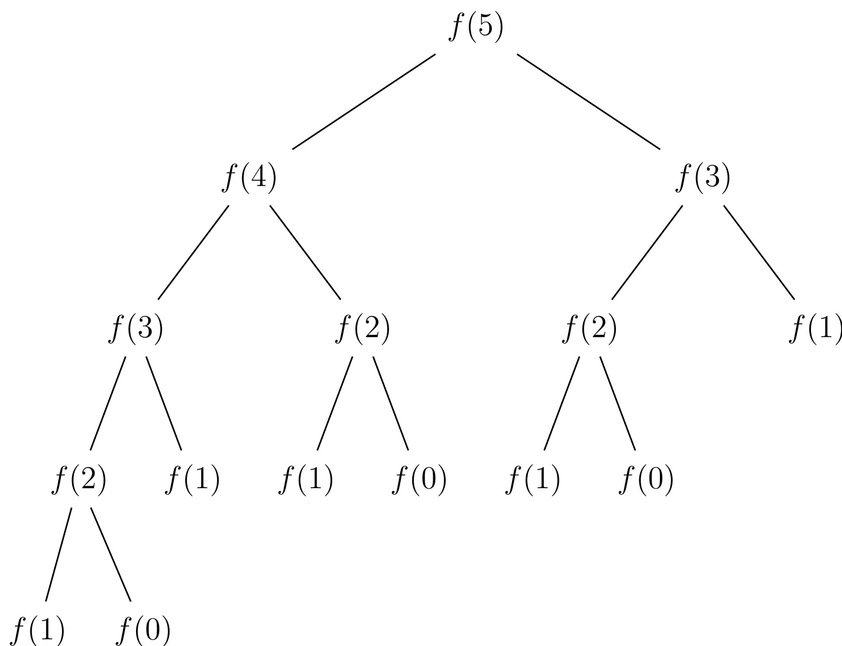

Abb. 32-1: Darstellung der rekursiven Aufrufe zur Berechnung der fünften Fibonacci-Zahl

Es lässt sich zeigen, dass die Zahl der Aufrufe der Methode `fibRekursiv(int)` exponentiell in dem Wert von n entwickelt: Es ist offensichtlich, dass die Zahl der Aufrufe der Methode `fibRekursiv(int)` ausgehend von dem Startwert n stets größer oder gleich der Zahl der Aufrufe der Methode `fibRekursiv(int)` ausgehend von dem Startwert $n - 1$ ist. Wenn wir die Zahl der Aufrufe von `fibRekursiv(int)` ausgehend von dem Startwert n als $a(n)$ bezeichnen, so können wir also (mit $n > 2$) berechnen:

$$a(n) = a(n-1) + a(n-2) \geq a(n-2) + a(n-2) = 2 \cdot a(n-2).$$

Der Wert $a(n)$ verdoppelt sich also jedes Mal wenn n um 2 vergrößert wird.

Allerdings wird man bei genauerer Inspektion des Aufrufbaums feststellen, dass die rekursive Lösung eine Menge doppelt berechnet: So wird beispielsweise der Wert $f(3)$ zwei Mal berechnet, der Wert von $f(2)$ gar drei Mal. Dynamische Programmierung ist nun ein Verfahren, das dazu dient, rekursive Lösungen im Hinblick auf die Laufzeit effizienter zu gestalten, indem man sich Werte, die einmal berechnet wurden, merkt, um sie bei Bedarf nicht erneut berechnen zu müssen.

Für die Fibonacci-Zahlen ergibt sich eine Lösung mittels dynamischer Programmierung beispielsweise wie folgt:

```
// Feld zur Speicherung der berechneten Fibonacci-Zahlen
private long[] berechneteFiboWerte;

public long fibDynProg(int n) {
       // Abfangen negativer Argumente:
       // Fibonacci-Zahlen sind nur für positive Zahlen definiert.
       if (n < 0) {
          throw new IllegalArgumentException();
       }
       // Initialisieren des Felds für bereits berechnete Werte
       berechneteFiboWerte = new long[n+1];
       // Berechnung der ersten n Fibonacci-Zahlen
       fibRekursivDyn(n);
       // Rückgabe der gesuchten Fibonacci-Zahl
       return berechneteFiboWerte[n];
}

private long fibRekursivDyn(int n) {
  // Rekursionsende: Die nullte und erste Fibonacci-Zahl sind 1.
  if (n == 0 || n == 1) {
    berechneteFiboWerte[n]=n;
    return n;
  }
  // Wenn die n-te Fibonacci-Zahl schon berechnet wurde, kann sie
  // einfach ausgelesen und zurückgegeben werden...
  if(berechneteFiboWerte[n] != 0)
      return berechneteFiboWerte[n];
  // ... anderenfalls muss sie rekursiv berechnet werden.
  return berechneteFiboWerte[n] =
      fibRekursivDyn(n-1) + fibRekursivDyn(n-2);
}
```

Durch diese Lösung verkleinert sich der Aufrufbaum signfikant, beispielsweise erhalten wir den folgenden Aufrufbaum für die Eingabe 5:

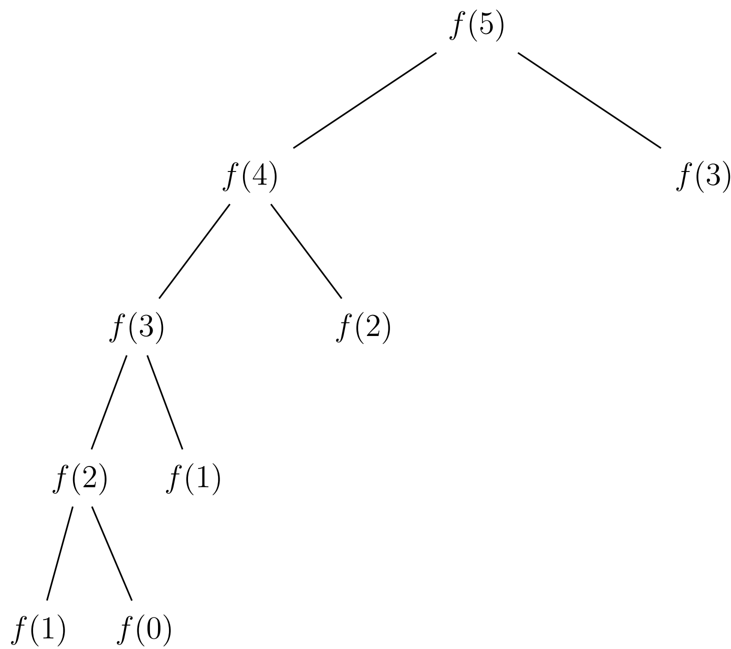

Abb. 32-2: Darstellung der rekursiven Aufrufe zur Berechnung der fünften Fibonacci-Zahl in der Lösung mit dynamischer Programmierung

Im Allgemeinen haben wir die Zahl der rekursiven Aufrufe von einem exponentiellen Wachstum im Argument n zu einem linearen Wachstum im Argument n reduziert. Im Gegenzug entsteht aber zusätzlicher Speicherbedarf, da das Feld mit den bereits berechneten Fibonacci-Zahlen verwaltet werden muss.

In vielen Fällen kann bei dem hier beschriebenen Vorgehen relativ einfach eine zusätzliche Optimierung vorgenommen werden, indem man die Lösung zu einer iterativen Lösung umformt. Im konkreten Fall der Fibonacci-Zahlen beispielsweise ist es schließlich so, dass zur Berechnung der n-ten Fibonacci-Zahl die Werte *aller* Fibonacci-Zahlen kleineren Indexes benötigt werden, man kann also den Overhead der Rekursion einsparen[2] und eine iterative Lösung wie die folgende wählen:

```
// Feld zur Speicherung der berechneten Fibonacci-Zahlen
private long[] berechneteFiboWerte;

public long fibDynProgIterativ(int n) {
  // Abfangen negativer Argumente:
  // Fibonacci-Zahlen sind nur für positive Zahlen definiert.
  if (n < 0) {
     throw new IllegalArgumentException();
  }
  // Initialisieren des Felds für bereits berechnete Werte
  berechneteFiboWerte = new long[n+1];
  // Die nullte und erste Fibonacci-Zahl sind 1.
  if(n < 2)
     return n;
```

2 Wir erinnern an den Stapel der Funktionsaufrufe, der im Speicher verwaltet werden muss.

```
  berechneteFiboWerte[0] = 0;
  berechneteFiboWerte[1] = 1;
  // Berechne die weiteren Fibonacci-Zahlen bis zur n-ten Zahl
  // von unten nach oben (bottom-up).
  for(int i=2; i<=n; ++i)
     berechneteFiboWerte[i] =
         berechneteFiboWerte[i-1] + berechneteFiboWerte[i-2];
  return berechneteFiboWerte[n];
}
```

Wir werden uns im Folgenden anschauen, wie ein Algorithmus mit Hilfe dynamischer Programmierung im Allgemeinen umgesetzt werden kann und in welchen Fällen es überhaupt sinnvoll ist, dynamische Programmierung zu verwenden. Obschon wir im einführenden Beispiel schlicht eine mathematische Funktion berechnet haben, wird dynamische Programmierung üblicherweise für Optimierungsprobleme eingesetzt. Aus diesem Grund werden wir uns in der Darstellung auf Optimierungsprobleme, wie sie gerade in wirtschaftlichen Anwendungen eine hohe Bedeutung haben, konzentrieren. Die Darstellungen und Bezeichnungen in diesem Kapitel folgen der Darstellung in

[CLRS10] Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, Robert L. Rivest:
Algorithmen – Eine Einführung
Oldenbourg, 3. Auflage, 2010

Dieses Buch sei an dieser Stelle jedem interessierten Leser, der mehr zum Design von Algorithmen lernen möchte, nahegelegt, da eine Vielzahl von Entwurfstechniken und Praxisbeispielen in anschaulicher Weise aufbereitet wurden. Das Buch ist sowohl im englischsprachigen Original (üblicherweise signifikant günstiger als die deutsche Übersetzung) als auch in der hier zitierten deutschen Übersetzung für alle Lernlevel geeignet.

32.1 Optimierungsprobleme und Anwendung der dynamischen Optimierung

Wir konzentrieren uns auf Optimierungsprobleme. Optimierungsprobleme sind Probleme, die mehrere verschiedene Lösungen haben, die jeweils einen zugehörigen Wert (beispielsweise eine natürliche Zahl oder eine reelle Zahl) haben. Das Ziel bei einem Optimierungsproblem ist, eine Lösung zu finden, die einen optimalen Wert hat, das heißt in der Regel eine Lösung die minimalen oder maximalen Wert besitzt. Wenn man beispielsweise Wege auf einer Landkarte betrachtet, die zwei Städte miteinander verbinden, könnte man daran interessiert sein, den (oder einen) Weg mit der minimalen Distanz zu identifizieren. Beim Vergleich von zwei DNA-Strängen ist ein wichtiges Kriterium die Länge der längsten gemeinsamen Teilsequenz. In diesem Fall gilt als optimaler Wert offensichtlich das Maximum (der Länge der übereinstimmenden Teilsequenz). Oftmals gibt es mehr als eine Lösung mit optimalem Wert, daher ist es genaugenommen nicht korrekt von *der* optimalen Lösung

zu sprechen, diese Feinheit soll im Laufe dieses Kurses aber keine besondere Bedeutung einnehmen.

Die Lösung eines Problems mittels dynamischer Programmierung erfolgt in einem vier-schrittigen Verfahren:

1. Charakterisierung der Struktur einer optimalen Lösung.
2. Rekursive Definition des Wertes einer optimalen Lösung.
3. Berechnung der optimalen Lösung, wahlweise mit einem iterativen (bottom-up) Verfahren, oder mit einem rekursiven Verfahren mit Speicherung der Zwischenergebnisse (sogenannte Memoisation).
4. Konstruktion der Optimallösung aus den Zwischenergebnissen (sofern eine optimale Lösung und nicht nur deren Wert erforderlich ist).

Wir demonstrieren diese vier wesentlichen Schritte anhand eines wirtschaftlichen Problems, des Stabzerlegungsproblems.

Definition 32.1-1: Stabzerlegungsproblem

Gegeben sei eine natürliche Zahl $n \in \mathbb{N}$, die wir Stablänge nennen und eine Zuordnung $p\colon \{1, 2, ..., n\} \to \mathbb{N}$, die jeder möglichen Stablänge (bis n) einen Preis zuordnet. Eine Lösung des Stabzerlegungsproblems ist eine Zuordnung $z\colon \{1, 2, ..., n\} \to \mathbb{N}_0$, so dass gilt:

$$\sum_{i=1}^{n} i \cdot z(i) = n.$$

Wir nennen dann z eine gültige Zerlegung. Der Wert einer Zerlegung z berechnet sich schließlich wie folgt:

$$e(z) = \sum_{i=1}^{n} z(i) \cdot p(i)$$

Wir nennen $e(z)$ den Erlös von z. Das Ziel des Stabzerlegungsproblems ist es nun, eine Zerlegung z zu finden, die den Erlös $e(z)$ maximiert. □

Informell kann das Problem wie folgt beschrieben werden: Wir besitzen eine Eisenstange der Länge n (beispielsweise in Zentimetern gemessen), die wir zu einem möglichst hohen Erlös verkaufen wollen. Hierzu holen wir uns auf dem freien Markt Angebote ein, wie hoch der Preis ist, den man für Eisenstangen unterschiedlicher Länge erzielen kann. Den Preis für Eisenstäbe unterschiedlicher Längen bezeichnen wir mit mit $p(1), p(2), ..., p(n)$.

Wir können nun unsere Eisenstange in einzelne Stücke zerschneiden und diese Einzelstücke dann verkaufen. Eine gültige Zerlegung z gibt also an, wie viele Stücke einer jeden Länge $1, 2, ..., n$ wir für den Verkauf herstellen. Dabei können wir natürlich nur eine Auswahl an Teilstücken treffen, die unsere ursprüngliche Eisenstange hergibt, sprich: Summieren wir die Länge aller Teilstücke auf, die wir abschneiden,

können wir die Länge der ursprünglichen Eisenstange nicht überschreiten; da wir keine negativen Verkaufspreise zulassen, können wir zudem davon ausgehen, dass es keinen Verschnitt gibt. Unseren Erlös $e(z)$ können wir schließlich berechnen, indem wir den Preis für jedes einzelne Stück Eisenstange, das wir hergestellt haben, aufaddieren.

Charakterisierung der Struktur einer optimalen Lösung Die Charakterisierung der Struktur einer optimalen Lösung erfolgt durch die Angabe einer Rekursionsformel, die den Wert der Optimallösung in Abhängigkeit von Zerlegungen des Problems in Teilprobleme angibt. In unserem Fall der Eisenstange können wir beispielsweise die folgende Formel verwenden:

$$o(n) = \mathsf{max}\{p(n), o(1) + o(n-1), o(2) + o(n-2), ..., o(n-1) + o(1)\} \qquad n \in \mathbb{N}$$

wobei $o(0) = 0$ definiert ist. Es ist hierbei zentral, dass eine Rückführung auf kleinere Probleminstanzen zur Bestimmung des Optimalwertes möglich ist. Die Idee der obigen Formel ist die folgende: Der höchste erzielbare Preis für eine Stange der Länge n ist das Maximum aus dem Preis, den man erzielen kann, wenn man die Stange unzerteilt verkauft ($p(n)$), und dem besten Preis einer jeden möglichen Zerlegung in zwei Teilstücke der Länge i, $n - i$. Man kann die Formel also verstehen als Auswahl, ob man die Stange lieber unzerteilt lassen sollte (das Maximum wird durch $p(n)$ erreicht), oder ob man eine Zerteilung an der Stelle i vornehmen sollte (das Maximum wird für $o(i)+o(n-i)$ erreicht). Rekursiv muss man im zweiten Fall aber noch entscheiden, ob man die Stücke der Länge i, $n - i$ noch weiter zerteilen möchte oder nicht.

Selbsttestaufgabe 32.1-1:

Wir betrachten das Problem der Stringteilung. Angenommen in einem Programm wird es häufig notwendig, eine Zeichenkette an mehreren Stellen zu zerschneiden. Gegeben sei also beispielsweise ein String der Länge 18*, sagen wir „Objektorientierung“ und wir wollen ihn an den Stellen* $2, 5, 12$ *zerteilen. Wir nehmen weiterhin an, dass bei jedem Zerteilungsschritt ein String entsteht, der den linken Teil und einen, der den rechten Teil des Ausgangsstrings enthält. Hierzu wird der String zeichenweise kopiert. Dann ist die Zahl der Kopierschritte abhängig von der Reihenfolge, in der man die Zerteilung vornimmt. Wenn wir beispielsweise von links nach rechts zerschneiden, erhalten wir folgende Ergebnisse:*

- *Im ersten Schritt erzeugen wir die Strings „Ob“ und „jektorientierung“; die Kosten für die Kopiervorgänge betragen* 18*, d.h. es sind* 18 *Zeichen zu kopieren.*
- *Im zweiten Schritt erzeugen wir die „jek“und „torientierung“; die Kosten für die Kopiervorgänge betragen* 16*, d.h. es sind* 16 *Zeichen zu kopieren.*
- *Im dritten Schritt erzeugen wir „torient“ und „ierung“; die Kosten für die Kopiervorgänge betragen* 13*, d.h. es sind* 13 *Zeichen zu kopieren.*

Insgesamt müssen $18 + 16 + 13 = 47$ *Zeichen kopiert werden. Bei umgekehrter Reihenfolge sind die Kosten wesentlich geringer, es müssen nur* $18 + 12 + 5 = 35$ *Zeichen kopiert werden.*
Charakterisieren Sie das Problem der Stringteilung rekursiv.

Rekursive Definition des Wertes einer optimalen Lösung Aufbauend auf der Rekursionsformel können wir nun eine Umsetzung des Optimalwertes als rekursives Programm vornehmen. In unserem Fallbeispiel der Zerteilung einer Eisenstange ergibt sich beispielsweise, wenn wir die Preisliste über ein Feld p von Ganzzahlen repräsentieren, bei dem der Index i den Preis für eine Eisenstange der Länge $i + 1$ angibt [3]:

```
public int eisenSchneidenRek(int n, int[] p) {
  // Eine Stange der Länge 0 hat keinen Wert
  if (n == 0)
     return 0;
  // Der Erlös der mit einer Stange der Länge n erzielt werden kann
  // ist mindestens so hoch wie der mit einer Stange der Länge n-1
  // erzielt werden kann.
  int optimum = p[n-1];
  // Überprüfe nacheinander die Erlöse aller möglicher Schnittstellen
  for(int i = 1; i<n; ++i){
      int aktuellerErloes = eisenSchneidenRek(i, p)
         + eisenSchneidenRek(n-i,p);
      if(aktuellerErloes > optimum)
        optimum = aktuellerErloes;
  }
  return optimum;
}
```

Selbsttestaufgabe 32.1-2:

Wir setzen das Beispiel der Stringteilung fort. Geben Sie eine rekursive Lösung des Problems (in Java) an.

Berechnung der optimalen Lösung Im vorherigen Schritt haben wir bereits ein Verfahren angegeben, das den Wert einer optimalen Lösung ermittelt, allerdings werden wie schon zuvor beim Berechnen der Fibonacci-Zahlen Werte wiederholt berechnet; wir können tatsächlich wiederum eine exponentielle Laufzeit nachweisen[4]. Daher wollen wir im Folgenden die bereits ermittelten Zwischenlösungen si-

3 Da Felder stets bei 0 beginnen, wir aber nur an Preisen ab der Länge 1 interessiert sind, nehmen wir diese Verschiebung vor.

4 Wir verzichten allerdings an dieser Stelle auf die Argumentation, da Komplexitätsanalyse nicht Kerninhalt dieses Kurses ist

chern, das kann mit einer direkten Modifikation unserer vorherigen Lösung ermöglicht werden:

```
// Methode, die die Memoisation initialisiert und vor dem Anwender
// verbirgt.
public int eisenSchneidenMemoised(int n, int[] p) {
  // Erzeugen des Feldes um die Optimalwerte zu speichern.
  int[] r = new int[n];
  return eisenSchneidenMemoisedRek(n, p, r);
}
```

```
private int eisenSchneidenMemoisedRek(int n, int[] p, int[] r) {
  // Falls der Wert bereits ermittelt wurde
  // kann er einfach ausgelesen werden.
  if (r[n-1] > 0)
     return r[n-1];
  // Rekursionsende: Eine Stang der Länge 0 hat keinen Wert.
  if (n == 0)
     return 0;
  // Mit einer Stange der Länge n kann man mindestens den Erlös
  // einer Stange der Länge n-1 erzielen.
  int optimum = p[n-1];
  // Teste rekursiv alle möglichen Schnittstellen.
  for(int i = 1; i<n; ++i){
      int aktuellerErloes =
        eisenSchneidenMemoisedRek(i, p, r)
        + eisenSchneidenMemoisedRek(n-i,p, r);
      if(aktuellerErloes > optimum)
        optimum = aktuellerErloes;
  }
  r[n-1] = optimum;
  return optimum;
}
```

Wir beobachten, dass wiederum für die Berechnung von $o(n)$ alle Werte $o(i)$ mit $i < n$ benötigt werden. Es ist also wieder möglich, dieses Programm ohne großen Aufwand in eine iterative Bottom-Up-Lösung umzubauen, die sich die Verwaltung rekursiver Aufrufe spart und dadurch sowohl weniger Speicher als auch weniger Rechenschritte benötigt als die gerade gefundene rekursive Methode mit Memoisation.

```
public int eisenSchneidenIte(int n, int[] p) {
  // Erzeugen des Feldes um die Optimalwerte zu speichern.
  int[] r = new int[n];
  // Ermittle den Optimalwert aufsteigend für alle Werte
  // kleiner gleich n.
  for(int j = 1; j <= n; ++j){
     int optimum = p[j-1];
```

```
      // Ermittle den Erlös durch Überprüfung aller möglichen
      // Schnittstellen.
      for(int i = 1; i < j; ++i){
         int aktuellerErloes = r[i-1] + r[j-i-1];
         if(aktuellerErloes > optimum)
            optimum = aktuellerErloes;
      }
   r[j-1]=optimum;
   }
   return r[n-1];
}
```

Selbsttestaufgabe 32.1-3:

Wir setzen das Beispiel der Stringteilung fort. Berechnen Sie die optimale Lösung mittels Memoisation oder mit Hilfe eines iterativen Bottom-Up-Verfahrens.
Hinweis: *Sie dürfen die folgende Methode* `hash(int[] z)` *verwenden, die einen Array von Zahlen von* 1 *bis* n *in eine Zahl von* 0 *bis* $2^n - 1$ *umformt:*

```
public int hash(int[] z) {
   int result = 0;
   for (int i=0; i < z.length; ++i) result += Math.pow(2,z[i]);
   return result;
}
```

Den Wert $2^{n+1}-1$ *können Sie in Java durch den Ausdruck* `(int)Math.pow(2,n+1)-1` *erhalten.*

Konstruktion der Optimallösung aus den Zwischenergebnissen Wir können zwar nun den optimalen Erlös ermitteln, den wir mit unserer Eisenstange erzielen können, doch dieses Ergebnis ist für sich genommen nicht befriedigend: Solange wir nicht wissen, wie wir unsere Eisenstange zerschneiden müssen um den optimalen Erlös zu erzielen, ist dieses Wissen von geringem praktischen Nutzen. Allerdings bedingt unser Vorgehen, dass wir bereits unterwegs eine Lösung implizit konstruieren, die den optimalen Wert besitzt, wir müssen also keine neuen Anstrengungen unternehmen, die gesuchte Lösung zu konstruieren, sondern müssen nur einen Weg finden, uns die Lösung unterwegs zu merken.

In unserem konkreten Beispiel der Eisenstangen bedeutet das, dass wir uns merken müssen, wo wir die Schnitte an dem jeweiligen Eisenstück setzen müssen, wenn wir den Maximalwert aktualisieren. Im Allgemeinen gilt: Immer wenn eine Entscheidung zu einer Verbesserung des bisherigen Optimalwerts führt, müssen wir uns merken, wie diese Entscheidung aussieht. Wir können insgesamt unser Programm wie folgt modifizieren:

```
public int eisenSchneidenIte(int n, int[] p, int[] s) {
  // Erzeugen des Feldes um die Optimalwerte zu speichern.
  int[] r = new int[n];
  // Ermittle den Optimalwert aufsteigend für alle Werte
  // kleiner gleich n.
  for(int j = 1; j <= n; ++j){
     int optimum = p[j-1];
     // Ermittle den Erlös durch Überprüfung aller möglichen
     // Schnittstellen.
     for(int i = 1; i < j; ++i){
        int aktuellerErloes = r[i-1] + r[j-i-1];
        if(aktuellerErloes > optimum){
           optimum = aktuellerErloes;
           // Merken der Schnittstelle, die den neuen
           // Optimalwert ergibt.
           s[j]=i;
        }
     }
  r[j-1]=optimum;
  }
  return r[n-1];
}
```

Mittels des Feldes `s` kann man nach Aufruf der Methode `eisenSchneidenIte(int, int[], int[])` eine Wahl, die Eisenstange zu schneiden rekonstruieren, die den maximalen Erlös garantiert. `s[n-1]` sagt dem Nutzer, an welcher Stelle die Eisenstange zunächst unterteilt werden soll. Die so erhaltenen Eisenstangen der Länge `s[n-1]` und $n -$ `s[n-1]` müssen nun potentiell wiederum zerteilt werden. An welcher Stelle die Eisenstangen zerteilt werden müssen, kann man im Feld s an den Postionen `s[n-1]` $- 1$ und $n -$ `s[n-1]` $- 1$ ablesen. Man beachte hier wiederum die Null-Indizierung des Feldes s.

Selbsttestaufgabe 32.1-4:

Wir setzen das Beispiel der Stringteilung fort. Modifizieren Sie Ihre Lösung zu Selbsttestaufgabe 32.1-3 so, dass die Lösung des Problems rekonstruiert werden kann.

32.2 Bedingungen für die Anwendbarkeit der dynamischen Programmierung

Dynamische Programmierung ist ein geeignetes Werkzeug um eine Vielzahl von (insbesondere Optimierungs-) Problemen ausgehend von einer rekursiven Charakterisierung zu lösen. Allerdings ist dynamische Programmierung wenig überraschend kein algorithmisches Allheilmittel. Es gibt zwei grundlegende Bedingungen,

die ein Problem erfüllen muss, damit dynamische Programmierung eine veritable Lösungsstrategie ist:

- Die Optimale Teilstruktur-Eigenschaft
- Überlappende Teilprobleme

Während die erste Eigenschaft eine notwendige Bedingung ist, damit dynamische Programmierung überhaupt ein korrektes Ergebnis liefert, ist die zweite Eigenschaft eine weichere Eigenschaft: Besitzt ein Problem keine überlappenden Teilprobleme, ist dynamische Programmierung grundsätzlich immernoch eine Strategie die zu einer korrekten Lösung führt, die Memoisation von Lösungen von Teilproblemen ist allerdings redundante Arbeit und die ursprüngliche rekursive Lösung wird durch die weiteren Arbeitsschritte der dynamischen Programmierung nicht mehr optimiert.

Wir werden im Folgenden die beiden Eigenschaften erläutern und an unserem Beispiel der Eisenstangen zeigen, wie man die Eigenschaften identifiziert, sowie jeweils ein Gegenbeispiel für ein Problem geben, das die jeweilige Eigenschaft *nicht* hat.

Optimale Teilstruktur Die optimale Teilstruktur-Eigenschaft bezeichnet die Eigenschaft, dass die Lösung eines Problems sich zusammensetzt aus den Lösungen kleinerer (gleichartiger) Probleminstanzen. Dieser Grundsatz hängt eng mit dem rekursiven algorithmischen Lösungsprinzip zusammen. Erinnern wir uns noch einmal an die drei Phasen eines rekursiven Algorithmus:

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.

Wenn man einen rekursiven Algorithmus zur Lösung eines Optimierungsproblems verwenden möchte, dann ist bereits implizit durch diese Struktur vorgegeben, dass es möglich sein muss, das Problem in Teilprobleme zu zerteilen und die Lösungen dieser Teilprobleme zu einer Lösung des Gesamtproblems zusammenzusetzen. Wenn ein Problem diese Eigenschaft nicht besitzt, so muss man sich wahlweise mit nicht optimalen Lösungen zufrieden geben oder aber einen anderen Lösungsansatz wählen.

Nun stellt sich natürlich die Frage, wie man die optimale Teilstruktur-Eigenschaft nachweist. Denn nur wenn einem dieser Nachweis gelingt, ist auch die Korrektheit des Verfahrens garantiert. Hierzu gibt es eine einfache Beweisstrategie, das Ausschneiden und Ersetzen. Ein solcher Beweis ist im Grunde genommen ein einfacher Widerspruchsbeweis, der wie folgt abläuft:

1. Wir nehmen zunächst an, dass die optimale Teilstruktur-Eigenschaft *nicht* erfüllt ist und führen diese Annahme in den weiteren Schritten zum Widerspruch.

2. Wir nehmen also an, dass unser Problem eine Instanz hat, zu der es eine Lösung gibt, bei der eine Teil-Probleminstanz nicht optimal gelöst ist.

3. Dann betrachten wir eine optimale Lösung dieser Teil-Probleminstanz. Wie wir an diese gelangen ist unerheblich, wichtig ist nur, dass es eine *echt bessere* Lösung dieses Teilproblems ist.

4. Wir ersetzen nun in der postulierten Optimallösung unserer ursprünglichen Probleminstanz die nicht optimale Lösung der Teilinstanz durch die optimale Lösung der Teilinstanz.

5. Wir zeigen schließlich, dass die so erhaltene Lösung korrekt ist...

6. und einen besseren Wert besitzt als die ursprünglich angenommene Lösung. Das steht jedoch im Widerspruch dazu, dass die ursprüngliche Lösung optimal war. Da die Wahl der Optimallösung beliebig war, kann es eine solche Optimallösung also nicht geben.

Zur Veranschaulichung beweisen wir die optimale Teilstruktur-Eigenschaft für das Stabzerlegungsproblem. Angenommen, das Stabzerlegungsproblem habe die optimale Teilstruktur-Eigenschaft nicht (1). Dann bedeutet das, dass es eine Stablänge n, eine Preiszuordnung p und eine gültige Zerlegung z gibt, so dass $e(z)$ maximal ist *und* es gibt eine Menge von Teilstäben deren Gesamtlänge i ist und deren Gesamterlös *geringer* ist als der größtmögliche Erlös für einen Stab der Länge i. Formal gibt es also eine Teilzerlegung $z'\colon \{1, 2, ..., i\} \to \mathbb{N}_0$ so dass $\sum_{j=1}^{i} j \cdot z'(j) = i$ (z' ist also eine gültige Zerlegung eines Stabes der Länge i) und $z'(j) \leq z(j)$ für alle $1 \leq j \leq i$ (z' ist also eine Teilzerlegung von z), so dass $e(z')$ nicht optimal ist (2). Das heißt, es gibt eine andere Zerlegung $z_i\colon \{1, 2, ..., i\} \to \mathbb{N}_0$ eines Stabes der Länge i mit $\sum_{j=1}^{i} j \cdot z_i(j) = i$, so dass $e(z_i) > e(z')$ (3).

Wir betrachten nun die folgende Zerlegung $\widehat{z}\colon \{1, 2, ..., n\} \to \mathbb{N}_0$ mit

$$\widehat{z}(j) = \begin{cases} z(j) - z'(j) + z_i(j) & j \leq i \\ z(j) & \text{sonst} \end{cases}.$$

Diese Zerlegung ersetzt also exakt die Teilzerlegung z' des Stabes der Länge i durch die ertragreichere Zerlegung z_i und lässt den Rest der Zerlegung unangetastet (4).

Dass $\widehat{z}$ eine legale Zerlegung ist (also tatsächlich eine Lösung des Problems) berechnet man sich wie folgt (5):

$$\sum_{j=1}^{n} j \cdot \widehat{z}(i) = \sum_{j=1}^{i} (z(j) - z'(j) + z_i(j)) + \sum_{j=i+1}^{n} z(j)$$
$$= \sum_{j=1}^{n} z(j) - \sum_{j=1}^{i} z'(j) + \sum_{j=1}^{i} z_i(j) = n - i + i = n$$

Dann berechnet sich der Ertrag $e(\widehat{z})$ wie folgt:

$$e(\widehat{z}) = \sum_{j=1}^{n} \widehat{z}(j) \cdot p(j) = \sum_{j=1}^{i} (z(j) - z'(j) + z_i(j)) \cdot p(j) + \sum_{j=i+1}^{n} \widehat{z}(j) \cdot p(j)$$

$$= \sum_{j=1}^{n} z(j) \cdot p(j) - \sum_{j=1}^{i} z'(j) \cdot p(j) + \sum_{j=1}^{i} z_i(j) \cdot p(j) = e(z) - e(z') + e(z_i) > e(z)$$

Das steht aber im Widerspruch zur Annahme, dass z eine optimale Zerlegung ist (6).

Die optimale Teilstruktur-Eigenschaft ist durchaus gelegentlich subtil, so dass eine genaue Überprüfung, ob ein Problem die optimale Teilstruktur-Eigenschaft aufweist, ratsam ist.

Ein Beispiel für ein Problem, das die optimale Teilstruktur-Eigenschaft nicht besitzt, können wir durch eine Zusatzregel für das Stabzerlegungsproblem finden. Angenommen es gäbe die Zusatzregel, dass jede Stablänge maximal ein Mal verkauft werden kann. Eine gültige Zerlegung z kann also nur den Wert 0 oder 1 für jede Länge i besitzen. Dann gilt die optimale Teilstruktur-Eigenschaft nicht, wie man an folgendem Beispiel leicht ersehen kann:

Wir betrachten einen Stab der Länge $n = 4$ und die Preiszuordnung $p(1) = 4$, $p(2) = 5$, $p(3) = 6$, $p(4) = 7$. Dann ist (wie man durch einfaches Ausprobieren einsehen kann) eine optimale Zerlegung der Gestalt, dass wir eine Stange der Länge 3 und eine Stange der Länge 1 wählen. Allerdings besitzt diese Zerlegung nicht die optimale Teilstruktur, denn die Stange der Länge 3 könnte für sich genommen in eine Stange der Länge 1 und eine Stange der Länge 2 zerlegt werden, mit einem Gesamterlös von $p(1) + p(2) = 4 + 5 = 9 > 6 = p(3)$. Diese Zerlegung der Stange der Länge 3 können wir aber nicht in unsere ursprüngliche Lösung einlegen, da wir anderenfalls zwei Stangen der Länge 1 herstellen würden und damit gegen die Regel verstoßen würden, dass jede Länge maximal ein Mal verkauft werden darf. Man beachte übrigens, dass ein Ansatz mit dynamischer Programmierung in diesem Fall nicht nur zu einem nicht optimalen, sondern sogar zu einem falschen Ergebnis führen würde.

Insbesondere globale Anforderungen an Lösungen (wie hier: Jede Stangenlänge darf nur ein Mal vorkommen) sorgen oft dafür, dass die optimale Teilstruktur-Eigenschaft nicht erfüllt ist und somit dynamische Programmierung nicht anwendbar ist. Der Grund hierfür ist, dass die Lösungen der Teilprobleme bei globalen Beschränkungen an die Lösung im Normalfall nicht unabhängig voneinander sind: Entscheidungen bei der Lösung eines Teilproblems können die Wahlmöglichkeiten bei der Lösung eines anderen Teilproblems einschränken. In diesem Sinne ist dynamische Programmierung also darauf angewiesen, dass sich ein Problem auf kleinere Teilprobleme zurückführen lässt, die sich ohne Kenntnis der Lösung der anderen Teilprobleme lösen lassen.

Selbsttestaufgabe 32.2-1:

Wir setzen das Beispiel der Stringteilung fort. Zeigen Sie, dass das Problem der Stringteilung die optimale Teilstruktur-Eigenschaft hat.

Überlappende Teilprobleme Die zweite Bedingung für die Anwendbarkeit von dymanischer Programmierung ist die, dass es überlappende Teilprobleme gibt. Das bedeutet, dass in der rekursiven Lösung des Problems mehrfach die gleiche Probleminstanz gelöst werden muss. Wir haben eingangs den Aufrufbaum einer rekursiven Lösung zur Berechnung der Fibonacci-Zahlen betrachtet und dabei beobachtet, dass beispielsweise zur rekursiven Berechnung der fünften Fibonacci-Zahl die zweite Fibonacci-Zahl drei Mal und die dritte Fibonacci-Zahl zwei Mal berechnet werden muss; die erste Fibonacci-Zahl muss sogar fünf Mal berechnet werden. An diesen Stellen treten also überlappende Teilprobleme auf.

Überlappende Teilprobleme können auch ohne weiteres bei dem Stabzerlegungsproblem beobachtet werden. Da die Redundanz bei diesem Problem aber sogar noch ausgeprägter ist, sei an dieser Stelle nur der Aufrufbaum für einen Stab der Länge 3 illustriert:

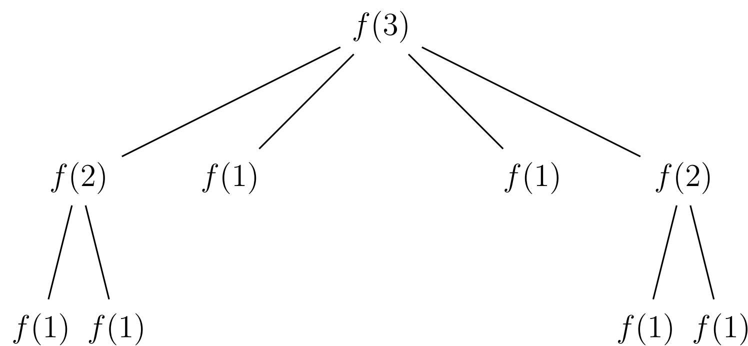

Abb. 32-3: Darstellung der rekursiven Aufrufe zur Berechnung des Stabzerlegungsproblems für die Länge $n = 3$

Wir sehen also, dass die rekursive Lösung sechs Mal das Teilproblem mit dem Index 1 lösen muss. Im Gegensatz zu der optimalen Teilstruktur-Eigenschaft ist diese Eigenschaft natürlich nicht notwendig, um mit dynamischer Programmierung ein korrektes Lösungsverfahren für ein Problem zu finden. Nichtsdestotrotz ist es sinnvoll, diese Bedingung zu stellen, denn ohne überlappende Teilprobleme kann man mit dynamischer Programmierung keine Performance-Steigerung im Vergleich zur rekursiven Lösung erzielen (die ohnehin im zweiten Arbeitsschritt erstellt wird) und erhöht somit nur ohne einen Gewinn den Speicherbedarf im Vergleich zur einfachen rekursiven Lösung. Wir haben bereits rekursive Algorithmen für Probleme kennen gelernt, die keine überlappenden Teilprobleme ergeben. Beispielsweise bei der rekursiven Berechnung der Fakultät oder dem Sortieren einer Liste (sowohl bei der Verwendung des rekursiven Verfahrens Quicksort, als auch Mergesort) entstehen keine überlappenden Teilprobleme.

Es ist übrigens zu beachten, dass die Eigenschaft, überlappende Teilprobleme zu besitzen, nicht eine inhärente Eigenschaft des Problems, sondern des rekursiven Lösungsverfahrens ist. Es kann mehr als ein rekursives Lösungsverfahren für ein gegebenes Problem geben, so dass ein Verfahren überlappende Teilprobleme generiert und ein anderes nicht.

Selbsttestaufgabe 32.2-2:

Wir setzen das Beispiel der Stringteilung fort. Zeigen Sie, dass das Problem der Stringteilung überlappende Teilprobleme besitzt.

33 Zusammenfassung

Mit Ausnahmen können wir auf irreguläres Nutzungsverhalten von Methoden hinweisen und beispielsweise unzulässige Argumente oder andere Fehlerzustände abfangen. Es ist möglich, geworfene Ausnahmen abzufangen und so von einem Fehlerzustand wieder in einen regulären Zustand überzugehen.

In dieser Kurseinheit haben wir sowohl die **lineare** als auch die **binäre Suche** kennen gelernt.

Für die **Sortierung** von Datenmengen kennen wir nun sowohl nicht rekursive Algorithmen, wie **Bubblesort**, **Sortieren beim Einfügen** und **Sortieren durch Auswählen** als auch rekursive Algorithmen, wie **Quicksort** und **Sortieren durch Verschmelzen**.

Rekursive Algorithmen bestehen aus **Basisfällen**, bei denen das Problem trivial lösbar ist, **rekursiven Aufrufen**, die das Problem verkleinern und es mit dem gleichen Verfahren lösen sowie der **Kombination der Teillösungen**. Wichtig dabei ist, dass eine Rekursion **konvergiert**.

Neben Suchen und Sortieren können rekursive Algorithmen auch bei vielen anderen Problemen angewendet werden, insbesondere bei verschiedenen Datenstrukturen.

Rekursive Lösungen lassen sich in vielen Fällen effizient mithilfe von dynamischer Programmierung umsetzen. Bedingungen hierfür sind **optimale Teilstruktur** und **überlappende Teilprobleme**.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 29.2-1:

```
void legeMehrwertsteuerFest(double neueMwSt) {
    if (neueMwSt < 0) {
        throw new IllegalArgumentException("Eine negative "
                      + "Mehrwertsteuer ist nicht zulaessig");
    }
    this.mehrwertsteuer = neueMwSt;
}
```

Die Ausnahme muss nicht im Methodenkopf angegeben werden.

Lösung zu Selbsttestaufgabe 29.3-1:

Bei einem Rabatt von `-0.2` wird die folgende Ausgabe erzeugt:

```
NegativerRabattAusnahme: Ein Rabatt von -0.2 ist nicht zulaessig.
```

Bei einem Rabatt von `1.2` wird die folgende Ausgabe erzeugt:

```
ZuHoherRabattAusnahme: Ein Rabatt von 1.2 ist nicht zulaessig.
```

Lösung zu Selbsttestaufgabe 29.3-2:

a. 1
b. 2
c. 5
d. 3
e. 4
f. 3
g. 4

Lösung zu Selbsttestaufgabe 29.3-3:

Bei der Anweisung `throw AnException();` in Methode `a` muss ein `new` ergänzt werden. Bei Methode `a` muss außerdem im Kopf `throws MyException, AnException` ergänzt werden. Es wäre auch möglich, stattdessen `throws Exception` zu ergänzen, jedoch sollten die Ausnahmetypen im Normalfall so genau wie möglich angegeben werden. Ein `try-catch`-Konstrukt ist in diesem Fall keine sinnvolle Lösung.

Bei Methode `b` kann ein `throws AnotherException` ergänzt werden. Dies ist aber nicht nötig, da es sich bei `AnotherException` um eine `RuntimeException` handelt.

In der Methode `callA()` müssen die Ausnahmen entweder gefangen werden, oder der Methodenkopf muss auch um eine `throws`-Anweisung ergänzt werden. Entscheidet man sich für `try-catch`, so sollte man zwei `catch`-Blöcke verwenden, statt nur einen für die gemeinsame Oberklasse `Exception`, um so jede Ausnahme gezielt behandeln zu können:

```
public void callA() {
    try {
        a(-2);
    } catch (MyException e) {
        System.out.println("MyException caught");
    } catch (AnException e) {
        System.out.println("AnException caught");
    }
}
```

Ließe man einen der beiden Blöcke weg, so würde dies einen Übersetzungsfehler verursachen.

Bei `callB()` kann man auch ein `try-catch` ergänzen oder eine `throws`-Anweisung im Kopf der Methode, muss dies jedoch nicht tun.

Lösung zu Selbsttestaufgabe 29.3-4:

Der Aufruf `z(-2)` erzeugt die Ausgabe `-1`, `z(0)` erzeugt die Ausgabe `-2`, und `z(7)` erzeugt die Ausgabe `49`.

Lösung zu Selbsttestaufgabe 30.1-1:

```
int bestimmeAnzahl(int wert, int[] feld) {
    if (feld == null) {
        return 0;
    }
    int anzahl = 0;
    for (int i : feld) {
        // Wert gefunden?
        if (i == wert) {
```

```
            anzahl++;
        }
    }
    return anzahl;
}
```

Lösung zu Selbsttestaufgabe 30.1-2:

Da uns hier die lexikalische Gleichheit zweier Zeichenketten interessiert (und nicht die Identität zweier Objekt-Verweise), sollten die Elemente mit Hilfe der Methode `equals()` (und nicht mit ==) verglichen werden. Die Methode `equals()` ist in der Klasse `String` so überschrieben, dass genau die Zeichenketten verglichen werden (vgl. Kapitel 26).

```
boolean istEnthalten(String wert, String[] feld) {
    if (feld == null) {
        return false;
    }
    for (String s : feld) {
        // Wert gefunden?
        if (s.equals(wert)) {
            return true;
        }
    }
    return false;
}
```

Lösung zu Selbsttestaufgabe 30.1-3:

```
int bestimmeGesamtbetragAllerRechnungenVon(Kunde k) {
    // wenn kein Kunde oder keine Rechnungen vorhanden
    if (k == null || rechnungen == null) {
        return 0;
    }
    int betrag = 0;
    // alle Rechnungen betrachten
    for (Rechnung r : rechnungen) {
        // wenn gesuchter Kunde
        if (r.liefereRechnungsempfaenger() == k) {
            // Betrag dazu addieren
            betrag += r.bestimmeBetragInCent();
        }
    }
    return betrag;
}
```

Lösung zu Selbsttestaufgabe 30.1-4:

```
Rechnung findeTeuersteRechnung() {
   // wenn keine Rechnungen vorhanden
    if (rechnungen == null) {
        return null;
    }
    Rechnung max = null;
    // alle Rechnungen betrachten
    for (Rechnung r : rechnungen) {
        // wenn noch kein Maximum gefunden
        // oder aktuelle Rechnung teurer
        if (max == null || max.bestimmeBetragInCent()
                          < r.bestimmeBetragInCent()) {
            // Maximum anpassen
            max = r;
        }
    }
    return max;
}
```

Lösung zu Selbsttestaufgabe 30.1-5:

```
    int bestimmeAnfangdesWorts(char[] feld, String wort) {
        for (int i = 0; i < feld.length - wort.length() + 1; i++) {
            for (int j = 0; j < wort.length(); j++) {
                // Vergleiche entsprechende Buchstaben
                if (feld[i + j] != wort.charAt(j)) {
                    // Buchstabe an Position j stimmt nicht überein
                    // suche bei i+1 weiter
                    break;
                } else if (j == wort.length() - 1) {
                    // gesamtes Wort wurde gefunden
                    return i;
                }
            }
        }
        // Wort konnte nicht vollständig gefunden werden
        return -1;
    }
```

Lösung zu Selbsttestaufgabe 30.2-1:

Es werden 18 Vergleiche und 12 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 30.2-2:

```
void sortiereAbsteigend(String[] feld) {
    // beginne beim zweiten Element und betrachte
    // die Liste bis zum Index i - 1 als sortiert
    for (int i = 1; i < feld.length; i++) {
        // gehe solange von i nach links
        // bis das Element an die richtige
        // Stelle gerutscht ist
        for (int j = i; j > 0; j--) {
            if (feld[j - 1].compareTo(feld[j]) < 0) {
                // wenn linkes kleiner,
                // vertausche die Elemente
                String temp = feld[j];
                feld[j] = feld[j - 1];
                feld[j - 1] = temp;
            } else {
                // sonst, ist das Element
                // an der richtigen Stelle
                break;
            }
        }
    }
}
```

Lösung zu Selbsttestaufgabe 30.2-3:

Es werden 25 Vergleiche und 12 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 30.2-4:

```
void sortiereAufsteigend(Rechnung[] rechnungen) {
    // es werden maximal length - 1 Durchläufe benötigt
    for (int i = 0; i < rechnungen.length - 1; i++) {
        // solange keine Vertauschungen vorgenommen werden
        // ist das Feld sortiert
        boolean sorted = true;
        // Durchlaufe das Feld, in jedem Durchlauf muss
        // ein Element weniger berücksichtigt werden
        for (int j = 0; j < rechnungen.length - 1 - i; j++) {
            if (rechnungen[j].bestimmeBetragInCent()
                    > rechnungen[j + 1].bestimmeBetragInCent()) {
                // wenn linkes größer dann vertausche
                Rechnung temp = rechnungen[j];
                rechnungen[j] = rechnungen[j + 1];
                rechnungen[j + 1] = temp;
                // Feld ist nicht sortiert
                sorted = false;
            }
        }
        if (sorted) {
```

```
                // keine Vertauschungen, Feld ist
                // folglich vollständig sortiert
                break;
            }
        }
    }
```

Lösung zu Selbsttestaufgabe 30.2-5:

Es werden 28 Vergleiche und 6 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 30.2-6:

```
void sortiere(double[] feld) {
    // suche length-1 mal das Maximum
    for (int i = 0; i < feld.length - 1; i++) {
        // Position des Maximums
        int max = 0;
        // Suche das Maximum
        for (int j = 1; j < feld.length - i; j++) {
            // wenn aktueller Wert größer als Maximum
            if (feld[j] > feld[max]) {
                // speichere aktuelle Position
                max = j;
            }
        }
        // setzte das Maximum an die letzte der unsortierten
        // Positionen, wenn es dort nicht schon ist
        if (max != feld.length - 1 - i) {
            double temp = feld[max];
            feld[max] = feld[feld.length - 1 - i];
            feld[feld.length - 1 - i]  = temp;
        }
    }
}
```

Lösung zu Selbsttestaufgabe 30.2-7:

Sortieren durch Einfügen:	
Vergleiche	**Vertauschungen**
18	12
7	0
28	28

Bubblesort:	
Vergleiche	**Vertauschungen**
25	12
7	0
28	28

Sortieren durch Auswählen:	
Vergleiche	**Vertauschungen**
28	6
28	0
28	4

Es fällt auf, dass Sortieren durch Auswählen weniger Vertauschungsoperationen als die anderen Verfahren benötigt, jedoch immer gleich viele Vergleiche. Die anderen beiden Verfahren schneiden dafür bei schon sortierten Feldern besser ab, sind jedoch bei umgekehrt sortierten Feldern deutlich langsamer.

Lösung zu Selbsttestaufgabe 31-1:

Es tritt ein `StackOverflowError` auf. Um dies zu verhindern, kann bei negativen Werten entweder eine `IllegalArgumentException` geworfen oder die Abfrage `n == 0` durch `n <= 0` ersetzt werden.

Lösung zu Selbsttestaufgabe 31-2:

```
int fakultaetIterativ(int n) {
    // ungültige Werte abfangen
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Ergebnis initialisieren
    int erg = 1;
    for (int i = 2; i <= n; i++) {
        erg *= i;
    }
    return erg;
}

int fakultaetRekursiv(int n) {
    // ungültige Werte abfangen
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Basisfall
    if (n == 0) {
        return 1;
    }
```

```
        // rekursiver Aufruf
        return n * fakultaetRekursiv(n - 1);
    }
```

Lösung zu Selbsttestaufgabe 31-3:

In jedem Schritt wird `n` um eins verringert. Somit nähert sich `n` immer weiter an den Basisfall `0` an.

Lösung zu Selbsttestaufgabe 31-4:

```
    public double power(int p, int n) {
        if (n < 0) {
    if (p==0)
    throw IllegalArgumentException("power(0,0) ist nicht definiert.");
            return 1.0 / power(p, -n);
        }
        if (n == 0) {
            return 1;
        }
        return p * power(p, n - 1);
    }
```

Lösung zu Selbsttestaufgabe 31-5:

0, 1, 1, 2, 3, 5, 8, 13, 21

Lösung zu Selbsttestaufgabe 31-6:

```
    long fibIterativ(int n) {
        if (n < 0) {
            throw new IllegalArgumentException();
        }
        // Basisfall: n ist 0 oder 1
        if (n == 0 || n == 1) {
            return n;
        }
        // die ersten beiden Zahlen
        int a = 0; // fib(0)
        int b = 1; // fib(1)
        // berechne die nächsten Zahlen
        for (int i = 2; i <= n; i++) {
            // fib(i) = fib(i - 1) + fib(i - 2);
            int temp = a + b;
            // Werte für nächste Berechnung
            // speichern
            a = b;
            b = temp;
```

```
    }
    return b;
}
```

Lösung zu Selbsttestaufgabe 31-7:

```
long zufallszahl(int n) {
    // Basisfall n < 3
    if (n < 3) {
        return n + 1;
    }
    // rekursive Aufrufe
    long f1 = zufallszahl(n - 1);
    long f2 = zufallszahl(n - 2);
    long f3 = zufallszahl(n - 3);
    // berechne Ergebnis
    return 1 + (((f1 - f2) * f3) % 100);
}

void gebeZufallszahlenAus() {
    for (int i = 5; i <= 30; i++) {
        System.out.println(zufallszahl(i));
    }
}
```

Lösung zu Selbsttestaufgabe 31-8:

```
boolean istPalindromIterativ(String s) {
    int laenge = s.length();
    // betrachte erste Haelfte des Wortes
    for (int i = 0; i < s.length() / 2; i++) {
        // und prüfe ob korrespondierendes
        // Zeichen identisch ist
        if (s.charAt(i) != s.charAt(laenge - 1 - i)) {
            return false;
        }
    }
    // alle Zeichen passen
    return true;
}
```

Lösung zu Selbsttestaufgabe 31-9:

```
boolean istPalindromRekursiv(String s) {
    // Basisfall
    if (s.length() <= 1) {
        return true;
    }
    // erstes und letztes Zeichen vergleichen
    if (s.charAt(0) != s.charAt(s.length() - 1)) {
        return false;
    }
    return istPalindromRekursiv(s.substring(
                                1, s.length() - 1));
}
```

Lösung zu Selbsttestaufgabe 31-10:

Bei der linearen Suche sind genau 11 Vergleiche nötig, bis der Wert gefunden wird.

Bei der binären Suche werden die folgenden Schritte und Vergleiche durchgeführt:

```
start =  0 und ende = 12 => mitte =  6 => Vergleich mit 27
start =  7 und ende = 12 => mitte =  9 => Vergleich mit 34
start = 10 und ende = 12 => mitte = 11 => Vergleich mit 40
start = 10 und ende = 10 => Basisfall  => Vergleich mit 38
```

Es finden insgesamt 4 Vergleiche mit den Werten `27, 34, 40` und `38` statt.

Lösung zu Selbsttestaufgabe 31-11:

```
    // wir gehen von einem sortierten Feld aus
    public boolean istEnthalten(String s, String[] feld)
        return istEnthalten(s, feld, 0, feld.length - 1);


// Wir gehen von einem sortierten Feld aus.
    private boolean istEnthalten(String s, String[] feld,
                                    int start, int ende) {
        // Basisfall: Bereich enthält maximal 2 Elemente
        if (ende - start <= 1) {
            return feld[start].equals(s) || feld[ende].equals(s);
        }
        // sonst: rekursive Aufteilung
        // Mitte bestimmen
        int mitte = start + (ende - start) / 2;
        System.out.println(mitte + " " + feld[mitte]);
        if (feld[mitte].equals(s)) {
            // wert gefunden
            return true;
        }
        if (feld[mitte].compareTo(s) > 0) {
            // in linker Hälfte suchen
```

```
            return istEnthalten(s, feld, start, mitte - 1);
        } else {
            // in rechter Hälfte suchen
            return istEnthalten(s, feld, mitte + 1, ende);
        }
    }
```

Lösung zu Selbsttestaufgabe 31-12:

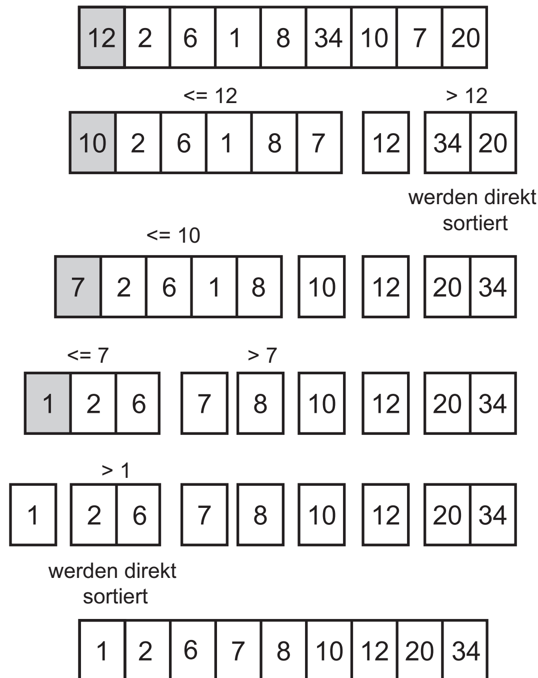

Abb. 31-8: Sortieren eines Beispielfeldes mit Quicksort

Lösung zu Selbsttestaufgabe 31-13:

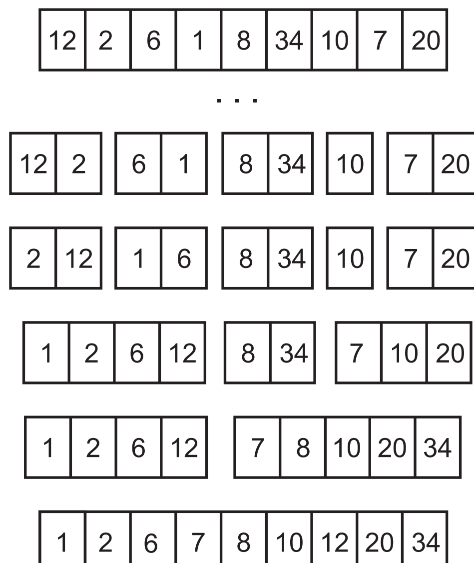

Abb. 31-11: Sortieren eines Beispielfeldes mit Sortieren durch Verschmelzen

Lösung zu Selbsttestaufgabe 32.1-1:

Es sei z ein Feld, das die Zerteilungsindizes angibt und n die Wortlänge des zu zerteilenden Worts, dann lassen sich die minimalen Kosten der Zerteilung $c(z, n)$ rekursiv wie folgt angeben:

- Falls z genau einen Eintrag hat, so sind die Kosten für die Zerteilung gerade n.
- Anderenfalls enthalte z $m > 1$ Einträge und wir schreiben $z[i..j]$ für das Teilfeld von z das genau die Einträge $i, i+1, ..., j$ enthält. Schließlich schreiben wir für ein Feld von natürlichen Zahlen a und eine natürliche Zahl x den Ausdruck $a - x$ um das Feld zu beschreiben, das man erhält, wenn man von jedem Eintrag von a den Wert x abzieht. Dann berechnet sich der Optimalwert gemäß

$$c(z, n) = n + \mathsf{min}\{c(z[0..i-1], z[i]) + c(z[i+1..m] - z[i], n - z[i]) \mid 0 \leq i \leq m\}$$

Lösung zu Selbsttestaufgabe 32.1-2:

```
public int c(int[] z, int n) {
    if(z.length == 1) return n;
    if(z.length == 0) return 0;
    int result = Integer.MAX_VALUE;
    for(int i = 0; i < z.length; ++i){
       int[] leftZ = new int[i];
       int[] rightZ = new int[z.length-i-1];
       for(int j=0; j<i;++j)
          leftZ[j]=z[j];
       for(int j=i+1; j<z.length; ++j)
          rightZ[j-i-1] = z[j]-z[i];
       int currentValue = n + c(leftZ, z[i])+ c(rightZ, n-z[i]);
       if(currentValue < result)
          result = currentValue;
    }
    return result;
}
```

Lösung zu Selbsttestaufgabe 32.1-3:

```
public int cMemoised(int[] z, int n) {
    int[][] savedValues = new int[n+1][(int)Math.pow(2,n+1)-1];
    return cMemoisedRek(z,n,savedValues);
}
public int cMemoisedRek(int[] z, int n, int[][] savedValues) {
    if(z.length == 1) {
       savedValues[n][hash(z)]=n;
       return n;
    }
    if(z.length == 0) return 0;
    int zHash = hash(z);
    if(savedValues[n][zHash]!=0) return savedValues[n][zHash];
    int result = Integer.MAX_VALUE;
    for(int i = 0; i < z.length; ++i){
       int[] leftZ = new int[i];
       int[] rightZ = new int[z.length-i-1];
       for(int j=0; j<i;++j)
          leftZ[j]=z[j];
       for(int j=i+1; j<z.length; ++j)
          rightZ[j-i-1] = z[j]-z[i];
       int currentValue = n + c(leftZ, z[i])+ c(rightZ, n-z[i]);
       if(currentValue < result)
          result = currentValue;
    }
    savedValues[n][zHash]=result;
    return result;
}
```

Lösung zu Selbsttestaufgabe 32.1-4:

```
public int cMemoised(int[] z, int n, int[][] choices) {
    int[][] savedValues = new int[n+1][(int)Math.pow(2,n+1)-1];
    return cMemoisedRek(z,n,savedValues);
}
public int cMemoisedRek(int[] z, int n, int[][] savedValues
   , int[][] choices) {
    if(z.length == 1) {
       savedValues[n][hash(z)]=n;
       choices[n][hash(z)]=z[0];
       return n;
    }
    if(z.length == 0) return 0;
    int zHash = hash(z);
    if(savedValues[n][zHash]!=0) return savedValues[n][zHash];
    int result = Integer.MAX_VALUE;
    for(int i = 0; i < z.length; ++i){
       int[] leftZ = new int[i];
       int[] rightZ = new int[z.length-i-1];
```

```
            for(int j=0; j<i;++j)
               leftZ[j]=z[j];
            for(int j=i+1; j<z.length; ++j)
               rightZ[j-i-1] = z[j]-z[i];
            int currentValue = n + c(leftZ, z[i])+ c(rightZ, n-z[i]);
            if(currentValue < result){
               result = currentValue;
               choices[n][hash(z)] = z[i];
            }
         }
         savedValues[n][zHash]=result;
         return result;
   }
```

Lösung zu Selbsttestaufgabe 32.2-1:

Angenommen das Stringteilungsproblem besäße die optimale Teilstruktur-Eigenschaft nicht. Dann gäbe es einen String der Länge n und Zerteilungsindizes z, so dass die optimale Reihenfolge der Zerteilungen von z einen Substring der Länge s generiert, der s gemäß z nicht optimal zerteilt wird, sagen wir es werden x Kopieroperationen benötigt, wenn man der Zerlegungsreihenfolge z folgt. Dann gibt es also eine Zerteilung z' dieses Substrings s, die weniger Kopieroperationen benötigt, sgen wir $y < x$. Einsetzen in die Rekursionsformel ergibt unmittelbar, dass Austauschen der Zerlegungsreihenfolge für s in z durch z' eine gültige Zerlegungsreihenfolge mit $c(z) - x + y < c(z)$ Kopieroperationen ergibt. Das steht im Widerspruch zur Annahme, dass z optimal gewählt ist.

Lösung zu Selbsttestaufgabe 32.2-2:

Dass das Stringteilungsproblem überlappende Teilprobleme besitzt, sieht man beispielsweise an der Eingabe $n = 8$, $z = [2, 4, 6]$. In diesem Fall muss das Teilproblem $n' = 6$, $z' = [2, 4]$ mehrfach gelöst werden.

Index

Algorithmus

- rekursiver, **338**

Ausnahme-Objekt, **319**

Ausnahmeklassen, **319**

Ausnahmesituation, **319**

Basisfall, **338**

binäre Suche, **346**

binary search, **346**

Bubblesort, **334**

catch, **324**

Error, **325**

Exception, **319**

Fakultätsfunktion, **340**

Fibonacci-Zahlen, **340**

finally, **324**

IllegalArgumentException, **322**

insertion sort, **333**

Interpolationsuche, **346**

Konvergenz, **338**

Laufzeitfehler, **319**

lineare Suche, **344**

merge sort, **349**

Methodenrahmen, **341**

Methodenstapel, **343**

Palindrom, **344**

Pseudozufallszahlen, **341**

Quicksort, **346**

Rekursion, **338**

Rekursive Definition, **338**

RuntimeException, **319**

selection sort, **336**

Sortieren, **332**

- beim Einfügen, **333**
- Bubblesort, **334**
- durch Auswählen, **336**
- durch Verschmelzen, **349**
- von Feldern, **332**, 346

StackOverflowError, **344**

String-Matching, **332**

Suchalgorithmen, **329**

Suchen, **329**

- auf mehrdimensionalen Feldern, **331**
- in Feldern, **329**
- in sortierten Feldern, **330**
- Maximum, **331**
- Minimum, **331**
- von Teilfolgen, **331**

throw-Anweisung, **321**

Throwable, **319**

throws, **321**

try-Anweisung, **324**