3 Rekursion

In der Natur und in der Mathematik findet man das Thema ***Selbstähnlichkeit*** bzw. sich wiederholende Strukturen, etwa für Schneeflocken oder Fraktale und Julia-Mengen, die interessante grafische Gebilde sind. Man spricht in diesem Zusammenhang von ***Rekursion***, und meint damit, dass Dinge sich wiederholen oder ähneln. Bezogen auf Methoden bedeutet dies, dass diese sich selbst aufrufen. Wichtig dabei ist eine Abbruchbedingung in Form spezieller Eingabewerte, die zum Ende der Selbstaufrufe führt.

3.1 Einführung

Diverse Berechnungen lassen sich hervorragend als rekursive Funktion beschreiben. Ziel dabei ist es, einen komplexeren Sachverhalt in mehrere einfachere Teilaufgabenstellungen herunterzubrechen.

3.1.1 Mathematische Beispiele

Nachfolgend schauen wir uns mit der Fakultät, der Summenbildung und den Fibonacci-Zahlen drei einführende Beispiele zur rekursiven Definition an.

Beispiel 1: Fakultät

Mathematisch ist die ***Fakultät*** für eine positive Zahl n als das Produkt (also die Multiplikation) aller natürlichen Zahlen von 1 bis einschließlich n definiert. Zur Notation wird das Ausrufezeichen der entsprechenden Zahl nachgestellt. Beispielsweise steht 5! für die Fakultät der Zahl 5:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Dies lässt sich wie folgt verallgemeinern:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1$$

Basierend darauf ergibt sich die rekursive Definition:

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

Dabei steht das umgedrehte »A« ($\forall$) für »für alle« .

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
n!	1	2	6	24	120	720	5040	40320

Berechnung der Fakultät in Java Schauen wir uns kurz an, wie sich die rekursive Berechnungsvorschrift der Fakultät in eine ebensolche Methode transferieren lässt:

```
public static int factorial(final int n)
{
    if (n < 0)
        throw new IllegalArgumentException("n must be >= 0");

    // rekursiver Abbruch
    if (n == 0 || n == 1)
        return 1;

    // rekursiver Abstieg
    return n * factorial(n - 1);
}
```

Verdeutlichen wir uns, was diese rekursive Definition an Aufrufen erzeugt:

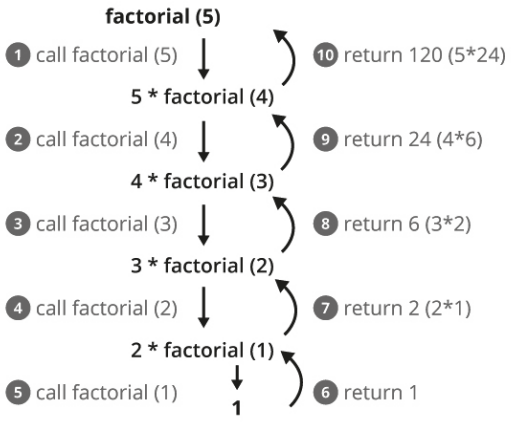

Abbildung 3-1 *Rekursive Aufrufe bei* `factorial(5)`

Beispiel 2: Berechnung der Summe der Zahlen bis n

Mathematisch ist die ***Summe*** für eine Zahl n als die Addition aller natürlichen Zahlen von 1 aufsteigend bis einschließlich n definiert:

$$\sum_{1}^{n} i = n + n - 1 + n - 2 + ... + 2 + 1$$

Das kann man folgendermaßen rekursiv definieren:

$$\sum_{1}^{n} i = \begin{cases} 1, & n = 1 \\ n + \sum_{1}^{n-1} i, & \forall n > 1 \end{cases}$$

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
sum(n)	1	3	6	10	15	21	28	36

Berechnung der Summe in Java Erneut überführen wir die rekursive Berechnungsvorschrift der Summation in eine rekursive Methode:

```
public static int sum(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be >= 1");

    // rekursiver Abbruch
    if (n == 1)
        return 1;

    // rekursiver Abstieg
    return n + sum(n - 1);
}
```

Achtung: Beschränkte Aufruftiefe

Bitte bedenken Sie, dass zur Summenbildung immer wieder Selbstaufrufe geschehen. Deswegen kann man hier nur einen Wert um die 10.000 – 20.000 übergeben. Bei größeren Werten kommt es zu einem `StackOverflowError`. Für andere rekursive Methoden gelten ähnliche Beschränkungen bezüglich der Anzahl an Selbstaufrufen.

Beispiel 3: Fibonacci-Zahlen

Auch die ***Fibonacci-Zahlen*** lassen sich hervorragend rekursiv definieren, wobei die Formel schon ein klein wenig komplexer ist:

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
fib(n)	1	1	2	3	5	8	13	21

Wenn man sich die Berechnungsvorschrift grafisch verdeutlicht, dann wird schnell klar, wie weit sich der Baum der Selbstaufrufe potenziell aufspannt – für größere n wäre der Aufrufbaum viel ausladender, wie es durch die gestrichelten Pfeile angedeutet ist:

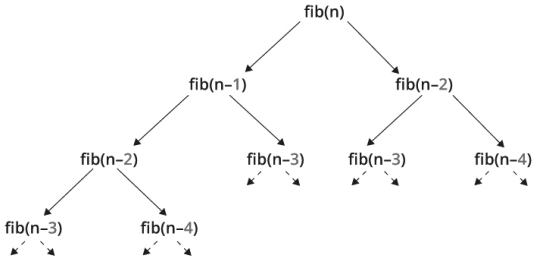

Abbildung 3-2 *Fibonacci rekursiv*

Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für $fib(n-4)$ und $fib(n-2)$, aber insbesondere dreimal für $fib(n-3)$. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Wie wir dies optimieren können, lernen wir später in Abschnitt 8.1 kennen.

Tipp: Abweichende Definition mit null als Startwert

Es sei noch angemerkt, dass es eine Abwandlung gibt, die beim Wert 0 startet. Dann gilt $fib(0) = 0$ und $fib(1) = 1$ und danach gemäß der rekursiven Definition $fib(n) = fib(n-1) + fib(n-2)$. Dies produziert die gleiche Zahlenfolge wie die obige Definition, nur um den Wert für die 0 ergänzt.

3.1.2 Algorithmische Beispiele

Zur Einführung haben wir uns mathematische Beispiele angeschaut. Darüber hinaus eignet sich Rekursion aber auch sehr gut für algorithmische Aufgabenstellungen, beispielsweise um für ein Array zu prüfen, ob die dort hinterlegten Werte ein Palindrom bilden. Unter einem Palindrom versteht man ein Wort, das sich von vorne und hinten gleich liest, etwa OTTO oder ABBA. Hier ist gemeint, dass die Elemente jeweils paarweise vorne und hinten übereinstimmen. Das gilt z. B. für ein `int[]` mit folgenden Werten: `{ 1, 2, 3, 2, 1 }`.

Beispiel 1: Palindrom – rekursive Variante

Die Prüfung auf Palindrom-Eigenschaft lässt sich rekursiv recht einfach lösen. Schauen wir uns dies als Programm an, nachdem ich kurz den Algorithmus beschrieben habe.

Algorithmus Wenn das Array die Länge 0 oder 1 hat, dann ist es per Definition ein Palindrom. Ist die Länge zwei und größer, dann wird jeweils das äußere linke und äußere rechte Element auf Übereinstimmung geprüft und danach eine Kopie des Arrays verkürzt um je eine Position vorne und hinten erzeugt. Die weitere Prüfung erfolgt im Anschluss auf dem verbliebenen Teilstück, wie es nachfolgendes Listing zeigt:

```
static boolean isPalindromeSimpleRecursive(final int[] values)
{
    // rekursiver Abbruch
    if (values.length <= 1)
        return true;

    int left = 0;
    int right = values.length - 1;

    if (values[left] == values[right])
    {
        // Achtung: copyOfRange, exklusive End
        final int[] remainder = Arrays.copyOfRange(values, left + 1, right);

        // rekursiver Abstieg
        return isPalindromeSimpleRecursive(remainder);
    }

    return false;
}
```

Das beschriebene und realisierte Vorgehen führt jedoch zu relativ vielen Kopien und Extraktionen von Teilarrays. Diesen Aufwand kann man vermeiden, wenn man zwar die Idee beibehält, den Algorithmus jedoch mithilfe eines Tricks minimal abwandelt.

Optimierter Algorithmus Statt der Kopien nutzt man weiterhin das Originalarray und verwendet zwei Positionsmarkierungen `left` und `right`, die initial das gesamte Array umfassen. Nun prüft man, ob der durch diese Positionen referenzierte linke und rechte Wert übereinstimmen. Ist das der Fall, verkleinert man den Prüfbereich auf beiden Seiten um eine Position und ruft das Ganze rekursiv auf. Das wird so lange wiederholt, bis der linke Positionszeiger den rechten überspringt.

Die Implementierung ändert sich wie folgt:

```
static boolean isPalindromeRecursive(final int[] values)
{
    return isPalindromeRecursive(values, 0, values.length - 1);
}

static boolean isPalindromeRecursive(final int[] values,
                                     final int left, final int right)
{
    // rekursiver Abbruch
    if (left >= right)
        return true;

    if (values[left] == values[right])
    {
        // rekursiver Abstieg
        return isPalindromeRecursive(values, left + 1, right - 1);
    }

    return false;
}
```

Vielleicht fragen Sie sich, wieso ich das Ganze nicht kompakter schreibe und weniger `return`-Anweisungen verwende. Bei der Darstellung dieser Algorithmen geht es mir vor allem um Verständlichkeit. Mehrere `return`s sind eigentlich nur dann ein Problem, wenn die Methode sehr lang und unübersichtlich ist.

Tipp: Hilfsmethoden zur Rekursionserleichterung

Die Idee von Positionszeigern in Arrays oder Strings ist bei Lösungen zur Rekursion ein gebräuchliches Mittel zur Optimierung und Vermeidung etwa von Array-Kopien. Damit das Ganze für Aufrufer nicht unkomfortabel wird, bietet sich eine High-Level-Methode an, die eine Hilfsmethode aufruft, die weitere Parameter besitzt. Dadurch ist es möglich, beim rekursiven Abstieg gewisse Informationen mitzugeben. In diesem Beispiel sind es die linke und rechte Grenze, sodass man auf potenziell aufwendige Kopien verzichten kann. Viele nachfolgende Beispiele werden von der generellen Idee Gebrauch machen.

Beispiel 1: Palindrom – iterative Variante

Obwohl eine rekursive Definition eines Algorithmus mitunter ziemlich elegant ist, produziert diese doch durch den rekursiven Abstieg in Form der Selbstaufrufe potenziell einiges an Overhead. Praktischerweise lässt sich jeder rekursive Algorithmus in einen iterativen umwandeln. Schauen wir uns dies für die Palindrom-Berechnung an. Für die iterative Umsetzung verwenden wir zwei Positionszeiger – statt des rekursiven Abstiegs nutzen wir eine `while`-Schleife. Diese bricht ab, wenn alle Elemente geprüft wurden oder falls zuvor eine Abweichung festgestellt wurde:

```
private static boolean isPalindromeIter(int[] values)
{
    int left = 0;
    int right = values.length - 1;
    boolean sameValue = true;

    while (left < right && sameValue)
    {
        sameValue = values[left] == values[right];

        left++;
        right--;
    }

    return sameValue;
}
```

Auch hier noch eine Anmerkung zur Kompaktheit: Diese Methode könnte man wie folgt schreiben, indem man auf die Hilfsvariable verzichtet:

```
static boolean isPalindromeIterativeCompact(final int[] values)
{
    int left = 0;
    int right = values.length - 1;

    while (left < right && values[left] == values[right])
    {
        left++;
        right--;
    }
    // left >= right || values[left] != values[right]
    return left >= right;
}
```

Der Rückgabewert ergibt sich aus der als Kommentar angedeuteten Bedingung, falls $left >= right$ gilt, dann ist `values` kein Palindrom. Bei dieser Variante muss man aber deutlich mehr über die Rückgabe nachdenken – ich bevorzuge erneut Verständlichkeit und Wartbarkeit gegenüber Kürzte oder Performance.

Beispiel 2: Fraktal als Beispiel

Wie eingangs erwähnt, lassen sich mit Rekursion auch Grafiken erzeugen. Nachfolgend wird eine grafisch simple Variante ausgegeben, die den Unterteilungen eines Lineals nachempfunden ist:

```
-
==
-
===
-
==
-
```

Tatsächlich lässt sich das unter Zuhilfenahme von Java-11-Bordmitteln (`repeat(n)`) leicht rekursiv wie folgt formulieren, wobei zweimal ein rekursiver Abstieg erfolgt:

```
static void fractalGenerator(final int n)
{
    if (n < 1)
       return;

    if (n == 1)
        System.out.println("-");
    else
    {
        fractalGenerator(n - 1);
        System.out.println("=".repeat(n));
        fractalGenerator(n - 1);
    }
}
```

Steht Java 11 und damit die Methode `repeat()` nicht zur Verfügung, dann schreibt man sich einfach eine Hilfsmethode `repeatCharSequence()` (vgl. Abschnitt 2.3.7).

Verwendet man statt ASCII-Zeichen etwas aufwendigere Zeichenfunktionen, so kann man mithilfe von Rekursion interessante und ansprechende Gebilde erzeugen, eingebettet in eine Swing-Applikation etwa folgende Schneeflocke.

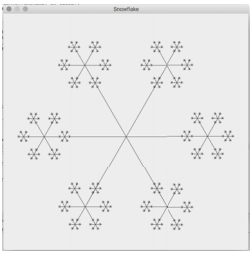

Abbildung 3-3 *Rekursive Grafik mit* `drawSnowflake()`

Diese stilisierte Darstellung einer Schneeflocke lässt sich wie folgt implementieren:

```
public static void drawSnowflake(final Graphics graphics,
                                 final int startX, final int startY,
                                 final int length, final int depth)
{
    for (int degree = 0; degree < 360; degree += 60)
    {
        final double rad = degree * Math.PI / 180;
        final int endX = (int) (startX + Math.cos(rad) * length);
        final int endY = (int) (startY + Math.sin(rad) * length);

        graphics.drawLine(startX, startY, endX, endY);

        // rekursiver Abstieg
        if (depth > 0)
        {
            drawSnowflake(graphics, endX, endY, length / 4, depth - 1);
        }
    }
}
```

3.1.3 Ablauf beim Multiplizieren der Ziffern einer Zahl

Zum Abschluss der algorithmischen Beispiele möchte ich nochmals die einzelnen Schritte und Selbstaufrufe verdeutlichen. Als artifizielles Beispiel nutzen wir dazu das Multiplizieren der Ziffern einer Zahl, auch Querprodukt genannt, etwa für den Wert $257 \Rightarrow 2 * 5 * 7 = 10 * 7 = 70$. Mithilfe von Modulo lassen sich die Extraktion der einzelnen Ziffern und deren Multiplikation recht einfach wie folgt implementieren:

```
static int multiplyAllDigits(final int value)
{
    final int remainder = value / 10;
    final int digitValue = value % 10;

    System.out.printf("multiplyAllDigits: %-10d | remainder: %d, digit: %d%n",
                      value, remainder, digitValue);

    if (remainder > 0)
    {
        final int result = multiplyAllDigits(remainder);

        System.out.printf("-> %d * %d = %d%n",
                          digitValue, result, digitValue * result);
        return digitValue * result;
    }
    else
    {
        System.out.println("-> " + value);
        return value;
    }
}
```

Betrachten wir die Ausgaben für die zwei Zahlen 1234 und 257:

```
jshell> multiplyAllDigits(1234)
multiplyAllDigits: 1234       | remainder: 123, digit: 4
multiplyAllDigits: 123        | remainder: 12, digit: 3
multiplyAllDigits: 12         | remainder: 1, digit: 2
multiplyAllDigits: 1          | remainder: 0, digit: 1
-> 1
-> 2 * 1 = 2
-> 3 * 2 = 6
-> 4 * 6 = 24
$2 ==> 24

jshell> multiplyAllDigits(257)
multiplyAllDigits: 257        | remainder: 25, digit: 7
multiplyAllDigits: 25         | remainder: 2, digit: 5
multiplyAllDigits: 2          | remainder: 0, digit: 2
-> 2
-> 5 * 2 = 10
-> 7 * 10 = 70
$3 ==> 70
```

Es ist gut ersichtlich, wie die rekursiven Aufrufe mit einer immer kürzeren Zahlenfolge geschehen und schließlich das Ergebnis auf Basis der letzten Ziffer in anderer Richtung konstruiert bzw. berechnet wird.

3.1.4 Typische Probleme

Rekursion erlaubt es oftmals, Problemstellungen auf verständliche Weise zu formulieren und zu implementieren. Dabei gibt es jedoch zwei Dinge zu beachten, auf die ich im Folgenden eingehe.

Endlose Aufrufe und `StackOverflowError`

Ein wissenswertes Detail ist, dass die Selbstaufrufe dazu führen, dass diese auf dem Stack zwischengespeichert werden. Für jeden Methodenaufruf wird ein sogenannter Stackframe mit Informationen zur aufgerufenen Methode und deren Parametern auf dem Stack abgelegt. Der Stack ist allerdings in seiner Größe beschränkt und somit können nur eine endliche Zahl an verschachtelten Methodenaufrufen erfolgen – in der Regel aber weit über 10.000. Das hatte ich bereits in einem Praxistipp kurz thematisiert.

Bei sehr vielen rekursiven Aufrufen kann es zu einem `StackOverflowError` kommen. Teilweise tritt die Problematik aber auch auf, weil keine Abbruchbedingung in der Rekursion vorgesehen wurde oder diese falsch formuliert ist:

```
// Achtung: Zur Demonstration bewusst falsch
static void infiniteRecursion(final String value)
{
    infiniteRecursion(value);
}

static int factorialNoAbortion(final int number)
{
    return number * factorialNoAbortion(number - 1);
}
```

Mitunter ist auch einfach der Aufruf falsch, weil kein verringerter Wert übergeben wird:

```
// Achtung: Zur Demonstration bewusst falsch
static int factorialWrongCall(final int n)
{
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;

    return n * factorialWrongCall(n);
}
```

Während man einen direkten endlosen Selbstaufruf noch recht gut optisch erkennen kann, wird dies mit zunehmender Anzahl Zeilen schwieriger: Die fehlende Abbruchbedingung in der Methode `factorialNoAbortion()` mag – spätestens mit etwas Erfahrung und Übung bezüglich Rekursion – auch noch gut erkennbar sein. In der Methode `factorialWrongCall()` ist das nicht mehr so leicht zu ermitteln. Hier muss man schon genauer wissen, wie die Logik sein sollte.

Aus den Beispielen sollten wir zwei Dinge mitnehmen:

1. **Abbruchbedingung** – Eine rekursive Methode muss immer auch mindestens einen Abbruch beinhalten. Aber selbst bei korrekter Definition kann es sein, dass beispielsweise der nicht erlaubte negative Wertebereich nicht überprüft wird. Für `factorial(int)` würde dann ein Aufruf mit einem negativen Wert zu einem `StackOverflowError` führen.
2. **Komplexitätsreduktion** – Eine rekursive Methode muss immer das ursprüngliche Problem in ein oder mehrere kleinere Teilprobleme untergliedern – manchmal ist dies bereits durch den um 1 reduzierten Wert eines Parameters gegeben.

Unerwartete Parameterwerte

Ein ziemlich fieser, weil leicht begangener und nur schwierig ersichtlicher Fehler kann bei Methodenaufrufen, insbesondere auch rekursiven, auftreten, wenn man Parameterwerte um eins erhöhen oder reduzieren möchte. Aus Gewohnheit ist man dazu verleitet, Postinkrement bzw. -dekrement (`++` bzw. `--`) zu verwenden, wie im folgenden Beispiel für die rekursive (leicht ungeschickte) Berechnung der Länge eines Strings, wobei der Parameter `count` die aktuelle Länge enthalten soll:

```
static int calcLengthParameterValues(final String value, int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);

    return calcLengthParameterValues(remaining, count++);
}
```

Beim Betrachten der Ausgaben für den Aufruf

```
final int length = calcLengthParameterValues("ABC", 0);
System.out.println("length: " + length);
```

sind wir vermutlich überrascht: Statt des um 1 erhöhten Werts, bleibt dieser gleich:

```
Count: 0
Count: 0
Count: 0
length: 0
```

Eine mögliche vorgegebene Maximalzahl kann somit nicht sichergestellt werden. In diesem Beispiel führt nur die Verkürzung der Eingabe zum Abbruch der Rekursion.

Interessanterweise lässt sich der Denkfehler schneller aufdecken, wenn man der guten Programmiertradition folgt und den Parameter `final` deklariert. Dadurch produziert der Compiler direkt eine Fehlermeldung und merkt an, dass die Variable nicht veränderlich ist. Daraufhin kommt man vielleicht eher auf die Idee, als Übergabe den Ausdruck `count + 1` zu wählen. Mit diesem Wissen fällt die Korrektur leicht:

```
static int calcLengthParameterValues(final String value, final int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);
    return calcLengthParameterValues(remaining, count + 1);
}
```

Gravierendere Auswirkungen Durch die gute Angewohnheit, Parameter als `final` zu definieren, lassen sich die vorgestellte Probleme entschärfen. Nehmen wir trotzdem einmal an, wir hätten bei der schon vorgestellten rekursiven Palindrom-Prüfung auf das `final` verzichtet und zudem etwas sorglos `++` bzw. `--` genutzt:

```
static boolean isPalindromeRecursive(int[] values, int left, int right)
{
    // rekursiver Abbruch
    if (left >= right)
        return true;

    if (values[left] == values[right])
    {
        // rekursiver Abstieg
        return isPalindromeRecursive(values, left++, right-);
    }
    return false;
}
```

Die Auswirkungen sind noch schlimmer als im vorherigen Beispiel, dass wenigstens terminiert, aber einen falschen Wert liefert. Bei der Palindrom-Prüfung erhalten wir stattdessen nach einiger Zeit einen `StackOverflowError`.

3.2 Aufgaben

3.2.1 Aufgabe 1: Fibonacci (★★☆☆☆)

Aufgabe 1a: Fibonacci rekursiv (★☆☆☆☆)

Schreiben Sie eine Methode `long fibRec(int)`, die die Fibonacci-Zahlen rekursiv basierend auf folgender Definition berechnet:

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Beispiel Prüfen Sie die Implementierung beispielsweise mit folgendem Werteverlauf:

Eingabe	1	2	3	4	5	6	7	8
fib(n)	1	1	2	3	5	8	13	21

Aufgabe 1b: Fibonacci iterativ (★★☆☆☆)

Die rekursive Berechnung der Fibonacci-Zahlen ist nicht effizient und die Laufzeit nimmt ab etwa der 40. – 50. Fibonacci-Zahl enorm zu. Schreiben Sie eine iterative Variante zur Berechnung: Was muss man dabei für die 1000. Fibonacci-Zahl beachten?

3.2.2 Aufgabe 2: Ziffern verarbeiten (★★☆☆☆)

Aufgabe 2a: Ziffern zählen (★★☆☆☆)

Schreiben Sie eine rekursive Methode `int calcDigits(int)`, die die Anzahl an Ziffern in einer positiven natürlichen Zahl ermittelt.

Aufgabe 2b: Quersumme (★★☆☆☆)

Berechnen Sie rekursiv die Quersumme einer Zahl. Diese ist definiert als die Summe ihrer Ziffern. Schreiben Sie dazu eine rekursive Methode `int calcSumOfDigits(int)`.

Beispiele

Eingabe	**Anzahl Ziffern**	**Quersumme**
1234	4	1 + 2 + 3 + 4 = 10
1234567	7	1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

3.3 Lösungen

3.3.1 Lösung 1: Fibonacci (★★☆☆☆)

Lösung 1a: Fibonacci rekursiv (★☆☆☆☆)

Schreiben Sie eine Methode `long fibRec(int)`, die die Fibonacci-Zahlen rekursiv basierend auf folgender Definition berechnet:

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Beispiel Prüfen Sie die Implementierung beispielsweise mit folgendem Werteverlauf:

Eingabe	1	2	3	4	5	6	7	8
fib(n)	1	1	2	3	5	8	13	21

Algorithmus Die Umsetzung in Java erfolgt exakt aus der mathematischen Definition:

```
static long fibRec(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be >= 1");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

    // rekursiver Abstieg
    return fibRec(n - 1) + fibRec(n - 2);
}
```

Lösung 1b: Fibonacci iterativ (★★☆☆☆)

Die rekursive Berechnung der Fibonacci-Zahlen ist nicht besonders effizient und die Laufzeit nimmt ab etwa der 40. – 50. Fibonacci-Zahl enorm zu. Schreiben Sie eine iterative Variante zur Berechnung: Was muss man dabei für die 1000. Fibonacci-Zahl beachten?

Algorithmus Ebenso wie die rekursive Variante prüft die iterative Umsetzung zunächst die Eingabe auf Gültigkeit und dann die Spezialfälle für den Aufruf mit den Werten 1 oder 2. Danach verwendet man zwei Hilfsvariablen und eine Schleife, die von 2 bis n läuft und dann zunächst die korrespondierende Fibonacci-Zahl aus der Summe der beiden Hilfsvariablen errechnet. Danach werden die beiden Hilfsvariablen passend zugewiesen. Damit ergibt sich folgende Implementierung:

```
static long fibIterative(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be >= 1");

    if (n==1 || n == 2)
        return 1;

    long fibN_2 = 1;
    long fibN_1 = 1;

    for (int count = 2; count < n; count++)
    {
        long fibN = fibN_1 + fibN_2;

        // um eins "weiterschieben"
        fibN_2 = fibN_1;
        fibN_1 = fibN;
    }

    return fibN;
}
```

Fibonacci für größere Zahlen Wenn Sie beispielsweise die 1000. Fibonacci-Zahl berechnen wollen, so reicht der Wertebereich eines `long` bei Weitem nicht mehr aus. Als Abhilfe muss die Berechnung mithilfe der Klasse `BigInteger` erfolgen.

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
@ParameterizedTest(name = "fibRec({0}) = {1}")
@CsvSource({ "1, 1", "2, 1", "3, 2", "4, 3", "5, 5", "6, 8", "7, 13", "8, 21" })
void fibRec(int n, long expectedFibN)
{
    long result1 = Ex01_Fibonacci.fibRec(n);

    assertEquals(expectedFibN, result1);
}

@ParameterizedTest(name = "fibIterative({0}) = {1}")
@CsvSource({ "1, 1", "2, 1", "3, 2", "4, 3", "5, 5", "6, 8", "7, 13", "8, 21" })
void fibIterative(int n, long expectedFibN)
{
    long result = Ex01_Fibonacci.fibIterative(n);

    assertEquals(expectedFibN, result);
}
```

3.3.2 Lösung 2: Ziffern verarbeiten (★★☆☆☆)

Lösung 2a: Ziffern zählen (★★☆☆☆)

Schreiben Sie eine rekursive Methode `int calcDigits(int)`, die die Anzahl an Ziffern in einer positiven natürlichen Zahl ermittelt. Wie man Ziffern extrahieren kann, haben wir bereits im vorherigen Kapitel in Abschnitt 2.1 besprochen.

Beispiele

Eingabe	Anzahl Ziffern	Quersumme
1234	4	1 + 2 + 3 + 4 = 10
1234567	7	1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

Algorithmus Wenn die Zahl kleiner als 10 ist, dann liefere den Wert 1, weil dies einer Ziffer entspricht. Ansonsten berechne den restlichen Wert, indem die Zahl durch 10 geteilt wird. Damit ruft man die Zählmethode rekursiv wie folgt auf:

```
static int calcDigits(final int value)
{
    if (value < 0)
        throw new IllegalArgumentException("value must be >= 0");

    // rekursiver Abbruch
    if (value < 10)
        return 1;

    final int remainder = value / 10;

    // rekursiver Abstieg
    return calcDigits(remainder) + 1;
}
```

Lösung 2b: Quersumme (★★☆☆☆)

Berechnen Sie rekursiv die Quersumme einer Zahl. Diese ist definiert als die Summe ihrer Ziffern. Schreiben Sie dazu eine rekursive Methode `int calcSumOfDigits(int)`.

Algorithmus Basierend auf der Lösung zur ersten Teilaufgabe variieren wir lediglich die Rückgabe bei der Ziffer sowie die Addition und den Selbstaufruf wie folgt:

```
static int calcSumOfDigits(final int value)
{
    if (value < 0)
        throw new IllegalArgumentException("value must be >= 0");

    // rekursiver Abbruch
    if (value < 10)
        return value;
```

```
    final int remainder = value / 10;
    final int lastDigit = value % 10;

    // rekursiver Abstieg
    return calcSumOfDigits(remainder) + lastDigit;
}
```

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
@ParameterizedTest(name = "calcDigits({0}) = {1}")
@CsvSource({ "1234, 4", "1234567, 7" })
void calcDigits(int number, int expected) throws Exception
{
    long result = Ex02_CalcDigits.calcDigits(number);

    assertEquals(expected, result);
}

@ParameterizedTest(name = "calcSumOfDigits({0}) = {1}")
@CsvSource({ "1234, 10", "1234567, 28" })
void calcSumOfDigits(int number, int expected) throws Exception
{
    long result = Ex02_CalcDigits.calcSumOfDigits(number);

    assertEquals(expected, result);
}
```

3.3.3 Lösung 3: ggT / GCD (★★☆☆☆)

Lösung 3a: ggT rekursiv (★☆☆☆☆)

Schreiben Sie eine Methode `int gcd(int, int)`, die den größten gemeinsamen Teiler (ggT) berechnet.[2] Im Englischen heißt dieser Greatest Common Divisor (GCD) und lässt sich mathematisch rekursiv wie folgt für zwei natürliche Zahlen a und b definieren:

$$gcd(a,b) = \begin{cases} a, & b = 0 \\ gcd(b, a \,\%\, b), & b \neq 0 \end{cases}$$

Beispiele

Eingabe 1	Eingabe 2	Resultat
42	7	7
42	28	14
42	14	14

[2] Umgangssprachlich ist damit die größte natürliche Zahl gemeint, durch die sich zwei ganze Zahlen ohne Rest teilen lassen.

8 Rekursion Advanced

In diesem Kapitel beschäftigen wir uns mit einigen fortgeschritteneren Aspekten rund um das Thema Rekursion. Wir starten mit der Optimierungstechnik namens Memoization. Im Anschluss schauen wir uns Backtracking als eine Problemlösungsstrategie an, die auf Versuch und Irrtum beruht und mögliche Lösungswege durchprobiert. Obwohl dies bezüglich der Performance nicht optimal ist, lassen sich damit diverse Implementierungen gut nachvollziehbar halten.

8.1 Memoization

In Kapitel 3 hatten wir bereits erkannt, dass sich mit Rekursion viele Algorithmen und Berechnungen verständlich und zugleich elegant beschreiben lassen. Allerdings war uns auch aufgefallen, dass Rekursion teilweise zu vielen Selbstaufrufen führt, die sich negativ auf die Performance auswirken können. Das gilt etwa für die Berechnung der Fibonacci-Zahlen oder die des Pascal'schen Dreiecks. Wie kann man dieses Problems Herr werden?

Dazu existiert eine nützliche Technik, die sich ***Memoization*** nennt. Diese verfolgt die gleichen Ideen wie das Caching oder Zwischenspeichern zuvor berechneter Werte. Man vermeidet dadurch mehrfache Ausführungen, indem bereits ermittelte Ergebnisse für Folgeaktionen wiederverwendet werden.

8.1.1 Memoization für Fibonacci-Zahlen

Praktischerweise kann man Memoization einem bestehenden Algorithmus oft leicht hinzufügen und muss diesen nur minimal anpassen. Das wollen wir für die Berechnung der Fibonacci-Zahlen nachvollziehen.

Wiederholen wir kurz die rekursive Definition der ***Fibonacci-Zahlen***:

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Die rekursive Umsetzung in Java folgt exakt der mathematischen Definition:

```
static long fibRec(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be > 0");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

    // rekursiver Abstieg
    return fibRec(n - 1) + fibRec(n - 2);
}
```

Wie fügt man nun Memoization hinzu? Tatsächlich ist dies nicht allzu schwierig. Wir benötigen eine Hilfsmethode, die die eigentliche Berechnungsmethode aufruft, sowie vor allem eine Datenstruktur zur Speicherung von Zwischenergebnissen. In diesem Fall nutzen wir eine Map, die an die Berechnungsmethode übergeben wird:

```
static long fibonacciOptimized(final int n)
{
    return fibonacciMemo(n, new HashMap<>());
}
```

In der ursprünglichen Methode umrahmen wir die eigentliche Berechnung mit den Aktionen zur Memoization: Bei jedem Berechnungsschritt schauen wir zuerst in der Map nach, ob bereits ein Ergebnis vorliegt, und geben dieses in dem Fall zurück. Ansonsten führen wir den Algorithmus wie zuvor aus, mit der minimalen Abwandlung, dass wir das Berechnungsergebnis in einer Variablen zwischenspeichern, um es zum Abschluss passend in der Lookup-Map hinterlegen zu können:

```
static long fibonacciMemo(final int n, final Map<Integer, Long> lookupMap)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be > 0");

    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis existiert
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = 0;
    // rekursiver Abbruch
    if (n == 1 || n == 2)
        result = 1;
    // rekursiver Abstieg
    else
        result = fibonacciMemo(n - 1, lookUpMap) +
                 fibonacciMemo(n - 2, lookUpMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);
    return result;
}
```

Performance-Vergleich Führt man die beiden Varianten für die 47. Fibonacci-Zahl aus, so liefert die rein rekursive Variante auf meinem iMac 4 GHz nach etwa 7 Sekunden ein Ergebnis, die andere mit Memoization dagegen nach wenigen Millisekunden.

Anmerkungen Es sei darauf hingewiesen, dass es eine Variante der Fibonacci-Berechnung gibt, die beim Wert 0 startet. Dann gilt $fib(0) = 0$ sowie $fib(1) = 1$ und danach rekursiv $fib(n) = fib(n-1) + fib(n-2)$. Das produziert die gleiche Zahlenfolge, wie die initiale Definition, nur um den Wert für die 0 ergänzt.

Weiterhin gibt es folgende Punkte zu bedenken:

- **Datentyp** – Die berechneten Fibonacci-Zahlen können ziemlich schnell sehr groß werden, sodass selbst der Wertebereich eines `long` nicht ausreicht und sich als Typ für die Rückgabe und die Lookup-Map ein `BigInteger` anbietet.
- **Rekursiver Abbruch** – Für die Implementierung ist es eine Überlegung wert, den rekursiven Abbruch vor der Verarbeitung mit Memoization zu notieren. Das wäre vermutlich minimal performanter, aber dann lässt sich der Algorithmus nicht so klar aus dem bestehenden umformen. Gerade wenn man mit Memoization noch nicht so vertraut ist, scheint die gezeigte Variante ein wenig eingängiger.

8.1.2 Memoization für Pascal'sches Dreieck

Das Pascal'sche Dreieck ist ebenso wie die Fibonacci-Zahlen rekursiv definiert:

$$pascal(row, col) = \begin{cases} 1, & row = 1 \text{ und } col = 1 \text{ (Spitze)} \\ 1, & \forall row \in \{1, n\} \text{ und } col = 1 \\ 1, & \forall row \in \{1, n\} \text{ und } col = row \\ pascal(row-1, col) + & \\ pascal(row-1, col-1), & \text{sonst (alle anderen Positionen)} \end{cases}$$

Schauen wir uns zunächst wieder die rein rekursive Implementierung an:

```
static int pascalRec(final int row, final int col)
{
    // rekursiver Abbruch: Spitze
    if (col == 1 && row == 1)
        return 1;

    // rekursiver Abbruch: Ränder
    if (col == 1 || col == row)
        return 1;

    // rekursiver Abstieg
    return pascalRec(row - 1, col) + pascalRec(row - 1, col - 1);
}
```

Auch für die Berechnung des Pascal'schen Dreiecks durch den Einsatz von Memoization ändert sich am ursprünglichen Algorithmus kaum etwas. Wir rahmen diesen lediglich mit den Zugriffen auf die Lookup-Map und die Speicherung dort ein:

```
static int pascalOptimized(final int row, final int col)
{
    return calcPascalMemo(row, col, new HashMap<>());
}

static int calcPascalMemo(final int row, final int col,
                          final Map<IntIntKey, Integer> lookupMap)
{
    // MEMOIZATION
    final IntIntKey key = IntIntKey.of(row, col);
    if (lookupMap.containsKey(key))
        return lookupMap.get(key);

    int result;
    // rekursiver Abbruch: Spitze
    if (col == 1 && row == 1)
        result = 1;
    // rekursiver Abbruch: Ränder
    else if (col == 1 || col == row)
        result = 1;
    else
        // rekursiver Abstieg
        result = calcPascalMemo(row - 1, col, lookupMap) +
                 calcPascalMemo(row - 1, col - 1, lookupMap);

    // MEMOIZATION
    lookupMap.put(key, result);
    return result;
}
```

Bei genauerem Hinsehen fällt uns auf, dass wir für den Key keinen Standardtyp, sondern durch die zweidimensionale Auslegung eine speziellere Variante, bestehend aus Zeile und Spalte, benötigen. Wir definieren dazu folgende Klasse `IntIntKey` – etwas Ähnliches wurde schon in der Lösung zur Aufgabe 7.3.12 ab Seite 350 entwickelt.

```
class IntIntKey
{
    final int value1;
    final int value2;

    public IntIntKey(final int value1, final int value2)
    {
        this.value1 = value1;
        this.value2 = value2;
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(value1, value2);
    }

    @Override
    public boolean equals(Object obj)
    {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
```

```
        IntIntKey other = (IntIntKey) obj;
        return value1 == other.value1 && value2 == other.value2;
    }

    @Override
    public String toString()
    {
        return "IntIntKey [value1=" + value1 + ", value2=" + value2 + "]";
    }
}
```

Vereinfachung mit Java 14 Mit Java 14 ist die Definition einfacher Datenbehälter-Klassen mithilfe des Schlüsselworts `record` mit der oben gezeigten Funktionalität wie folgt möglich:

```
static record IntIntKey(int value1, int value2)
{
}
```

Performance-Vergleich Um die Performance zu vergleichen, wählen wir einen Aufruf mit den Parametern für Zeile 42 und Spalte 15. Führt man diesen mit beiden Varianten aus, so benötigt die erste, rein rekursive Variante für die gewählten Werte auf einem iMac mit 4 Ghz eine ziemlich lange Laufzeit von rund 80 Sekunden, die optimierte Variante ist dagegen nach wenigen Millisekunden fertig.

Fazit

Für die beiden hier vorgestellten Beispiele führt die rein rekursive Definition jeweils zu einer Vielzahl an Selbstaufrufen und diese resultieren ohne Memoization darin, dass immer wieder die gleichen Zwischenergebnisse berechnet und verworfen werden. Das ist unnötig und kostet Performance.

Memoization ist als Abhilfe so einfach wie genial und effizient. Zudem kann man dadurch viele Probleme weiterhin elegant mithilfe eines rekursiven Algorithmus lösen, ohne jedoch die Nachteile bezüglich der Performance in Kauf nehmen zu müssen. Durch Memoization lässt sich die Laufzeit oftmals (sehr) deutlich verringern.

8.3 Aufgaben

8.3.1 Aufgabe 1: Türme von Hanoi (★★★☆☆)

Beim Türme-von-Hanoi-Problem gibt es drei Türme oder Stäbe A, B und C. Zu Beginn sind mehrere gelochte Scheiben der Größe nach auf Stab A platziert, die größte zuunterst. Ziel ist es nun, den gesamten Stapel, also alle Scheiben, von A nach C zu bewegen. Dabei darf immer nur eine Scheibe nach der anderen bewegt werden und niemals eine kleinere Scheibe unter einer größeren liegen. Deswegen benötigt man den Hilfsstab B. Schreiben Sie eine Methode `void solveTowersOfHanoi(int)`, die die Lösung auf der Konsole in Form der auszuführenden Bewegungen ausgibt.

Beispiel Das Ganze sieht in etwa wie folgt aus:

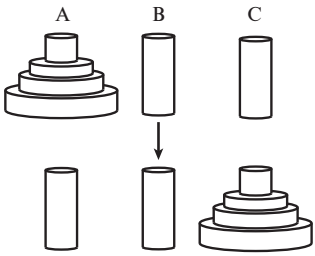

Abbildung 8-1 *Aufgabenstellung beim Türme-von-Hanoi-Problem*

Für drei Scheiben soll der folgende Lösungsweg ausgegeben werden:

```
Tower Of Hanoi 3
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Bonus Erstellen Sie eine konsolenbasierte grafische Aufbereitung. Für zwei Scheiben würde dies etwa wie folgt aussehen:

```
Tower Of Hanoi 2
   A         B         C
   |         |         |
  #|#        |         |
 ##|##       |         |
-------   -------   -------
Moving slice 1: Tower [A] -> Tower [B]
   A         B         C
   |         |         |
   |         |         |
 ##|##      #|#        |
-------   -------   -------
```

```
Moving slice 2: Tower [A] -> Tower [C]
   A         B         C
   |         |         |
   |         |         |
   |        #|#      ##|##
-------   -------   -------
Moving slice 1: Tower [B] -> Tower [C]
   A         B         C
   |         |         |
   |         |        #|#
   |         |       ##|##
-------   -------   -------
```

8.3.2 Aufgabe 2: Edit Distance (★★★★☆)

Berechnen Sie für zwei Strings, wie viele Änderungen diese – ohne Beachtung der Groß- und Kleinschreibung – auseinander liegen, also wie man den einen String in den anderen durch ein- oder mehrmaliges Anwenden einer beliebigen der folgenden Aktionen überführen kann:

- Ein Zeichen hinzufügen ($+$),
- ein Zeichen löschen ($-$) oder
- ein Zeichen verändern ($\rightsquigarrow$).

Schreiben Sie eine Methode `int editDistance(String, String)`, die zeichenweise die drei Aktionen testet und den anderen Teil rekursiv prüft.

Beispiele Für die gezeigten Eingaben sind folgende Modifikationen nötig:

Eingabe 1	Eingabe 2	Resultat	Aktionen
"Micha"	"Michael"	2	$\text{Micha} \underbrace{\rightarrow}_{+\text{e}} \text{Michae} \underbrace{\rightarrow}_{+\text{l}} \text{Michael}$
"Ananas"	"Banane"	3	$\text{Ananas} \underbrace{\rightarrow}_{+\text{B}} \text{BAnanas} \underbrace{\rightarrow}_{-\text{s}} \text{BAnana} \underbrace{\rightarrow}_{\text{a}\rightsquigarrow\text{e}} \text{BAnane}$

Bonus (★★★☆☆) Optimieren Sie Edit Distance mit Memoization

8.3.3 Aufgabe 3: Longest Common Subsequence (★★★☆☆)

In der vorherigen Aufgabe ging es darum, wie viele Änderungen benötigt werden, um zwei gegebene Strings ineinander zu überführen. Eine weiteres interessantes Problem besteht darin, die längste gemeinsame, aber nicht unbedingt zusammenhängende Sequenz von Buchstaben zu ermitteln, die in zwei Strings in der gleichen Abfolge auftritt. Schreiben Sie eine Methode `String lcs(String, String)`, die rekursiv von hinten die Strings verarbeitet und bei zwei gleich langen Teilstücken das zweite nutzt.

8.4 Lösungen

8.4.1 Lösung 1: Türme von Hanoi (★★★☆☆)

Beim Türme-von-Hanoi-Problem gibt es drei Türme oder Stäbe A, B und C. Zu Beginn sind mehrere gelochte Scheiben der Größe nach auf Stab A platziert, die größte zuunterst. Ziel ist es nun, den gesamten Stapel, also alle Scheiben, von A nach C zu bewegen. Dabei darf immer nur eine Scheibe nach der anderen bewegt werden und niemals eine kleinere Scheibe unter einer größeren liegen. Deswegen benötigt man den Hilfsstab B. Schreiben Sie eine Methode `void solveTowersOfHanoi(int)`, die die Lösung auf der Konsole in Form der auszuführenden Bewegungen ausgibt.

Beispiel Das Ganze sieht in etwa wie folgt aus:

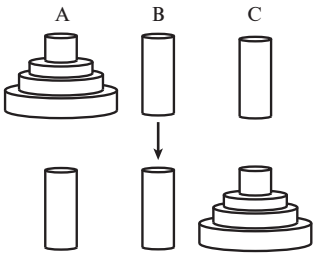

Abbildung 8-2 *Aufgabenstellung beim Türme-von-Hanoi-Problem*

Für drei Scheiben soll der folgende Lösungsweg ausgegeben werden:

```
Tower Of Hanoi 3
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Algorithmus Das Bewegen der Scheiben implementieren wir in einer Methode `void moveTower(int, char, char, char)`, der man die Anzahl der zu bewegenden Scheiben, den Ausgangsstab, den Hilfsstab und den Zielstab übergibt. Initial nutzt man n und 'A', 'B' sowie 'C' als Startparameter. Die Methode `moveTower()` zerteilt das Problem in drei kleinere Probleme:

1. Zunächst wird der um eine Scheibe kleinere Turm von der Ausgangsbasis auf den Hilfsstab transportiert.
2. Dann wird die letzte und größte Scheibe von der Quelle zum Ziel bewegt.
3. Zum Abschluss muss noch der restliche Turm vom Hilfsstab auf die Zielscheibe bewegt werden.

Als rekursiver Abbruch dient die Aktion »bewege Quelle nach Ziel«, wenn die Höhe 1 ist. Ein bisschen knifflig wird das Ganze durch das Vertauschen von Quelle, Ziel und Hilfsstab während der Aktionen.

```
void moveTower(final int n, final char source,
               final char helper, final char destination)
{
    if (n == 1)
        System.out.println(source + " -> " + destination);
    else
    {
        // bewege alle bis auf letzte Scheibe von Quelle auf Hilfsstab
        // Destination wird so zum neuen Hilfsstab
        moveTower(n - 1, source, destination, helper);

        // bewege die größte Scheibe von Quelle zum Ziel
        moveTower(1, source, helper, destination);

        // von Hilfsstab auf Ziel
        moveTower(n - 1, helper, source, destination);
    }
}
```

Damit man als Nutzer weniger Details sieht, bietet sich die Definition der folgenden Methode an:

```
void solveTowersOfHanoi(final int n)
{
    System.out.println("Tower Of Hanoi " + n);
    moveTower(n, 'A', 'B', 'C');
}
```

Zur Lösung des Problems muss die Methode mit der gewünschten Anzahl an Scheiben aufgerufen werden, etwa wie folgt:

```
jshell> solveTowersOfHanoi(3)
Tower Of Hanoi 3
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Tipp: Rekursion als Hilfsmittel

Obwohl das Problem zunächst ziemlich verzwickt klingt, lässt es sich mithilfe von Rekursion doch recht einfach lösen. An dieser Aufgabe wird nochmal deutlich, dass man Rekursion zur Reduktion der Schwierigkeit nutzt, indem ein Problem in mehrere kleinere, nicht so schwierig zu lösende Teilprobleme zerlegt wird.

Bonus: Erstellen Sie eine konsolenbasierte grafische Aufbereitung

Für zwei Scheiben könnte dies folgendermaßen aussehen:

```
Tower Of Hanoi 2
   A         B         C
   |         |         |
  #|#        |         |
 ##|##       |         |
-------   -------   -------
Moving slice 1: Tower [A] -> Tower [B]
   A         B         C
   |         |         |
   |         |         |
 ##|##      #|#        |
-------   -------   -------
Moving slice 2: Tower [A] -> Tower [C]
   A         B         C
   |         |         |
   |         |         |
   |        #|#      ##|##
-------   -------   -------
Moving slice 1: Tower [B] -> Tower [C]
   A         B         C
   |         |         |
   |         |        #|#
   |         |       ##|##
-------   -------   -------
```

Betrachten wir zunächst, wie sich der Algorithmus für grafische Ausgaben ändert: Tatsächlich bleibt dieser zur Lösungsfindung absolut gleich. Wir fügen unserer Implementierung eine Klasse `Tower` sowie eine Aktion, die wir beim Lösen in Form eines Lambdas mitgeben, hinzu. Die Methode `solveTowersOfHanoi(int)` wandeln wir derart ab, dass dort drei `Tower`-Objekte erzeugt werden und entsprechend auf dem Ausgangsturm die gewünschte Anzahl an Scheiben platziert wird:

```
void solveTowersOfHanoi(final int n)
{
    System.out.println("Tower Of Hanoi " + n);

    final Tower source = new Tower("A");
    final Tower helper = new Tower("B");
    final Tower destination = new Tower("C");

    // Achtung umgekehrte Reihenfolge: größte Scheibe zuerst
    for (int i = n; i > 0; i--)
        source.push(i);

    final Runnable action =
                  () -> printTowers(n + 1, source, helper, destination);
    action.run();

    moveTower(n, source, helper, destination, action);
}
```

Die Realisierung von `moveTower()` erhält lediglich ein `Runnable` als weiteren Parameter. Damit kann eine Aktion beim rekursiven Ende ausgeführt werden:

```
void moveTower(final int n, final Tower source, final Tower helper,
               final Tower destination, final Runnable action)
{
    if (n == 1)
    {
        final Integer elemToMove = source.pop();
        destination.push(elemToMove);

        System.out.println("Moving slice " + elementToMove +
                           ": " + source + " -> " + destination);
        action.run();
    }
    else
    {
        moveTower(n - 1, source, destination, helper, action);
        moveTower(1, source, helper, destination, action);
        moveTower(n - 1, helper, source, destination, action);
    }
}
```

Die Klasse `Tower` Machen wir uns daran, die Klasse `Tower` zu erstellen, die zur Identifikation einen String und zur Speicherung der Scheiben einen `Stack<E>` nutzt:

```
class Tower
{
    private final String name;
    private final Stack<Integer> values = new Stack<>();

    public Tower(final String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return "Tower [" + name + "]";
    }

    public void push(final Integer item)
    {
        values.push(item);
    }

    public Integer pop()
    {
        return values.pop();
    }

    ...
```

Konsolenausgabe der Türme In Kapitel 4 zu Strings haben wir in Abschnitt 4.2.16 in Aufgabe 16 eine erste Variante zum Zeichnen von Türmen kennengelernt. Das dort gewonnene Wissen nutzen wir und wandeln die Implementierung geeignet ab. Zunächst zeichnen wir den oberen Teil des Stabs mit `drawTop()`. Danach zeichnen wir die Scheiben mit `drawSlices()` und schließlich eine untere Begrenzungslinie mit `drawBottom()`:

```
static List<String> printTower(final int maxHeight)
{
    final int height = values.size() - 1;

    final List<String> visual = new ArrayList<>();

    visual.addAll(drawTop(maxHeight, height));
    visual.addAll(drawSlices(maxHeight, height));
    visual.add(drawBottom(maxHeight));

    return visual;
}

private List<String> drawTop(final int maxHeight, final int height)
{
    final List<String> visual = new ArrayList<>();
    final String nameLine = repeatCharSequence(" ", maxHeight) + name +
                            repeatCharSequence(" ", maxHeight);
    visual.add(nameLine);

    for (int i = maxHeight - height - 1; i > 0; i--)
    {
        final String line = repeatCharSequence(" ", maxHeight) + "|" +
                            repeatCharSequence(" ", maxHeight);
        visual.add(line);
    }
    return visual;
}

static List<String> drawSlices(final int maxHeight, final int height)
{
    final List<String> visual = new ArrayList<>();

    for (int i = height; i >= 0; i--)
    {
        final int value = values.get(i);
        final int padding = maxHeight - value;

        final String line = repeatCharSequence(" ", padding) +
                            repeatCharSequence("#", value) + "|" +
                            repeatCharSequence("#", value);
        visual.add(line);
    }

    return visual;
}

static String drawBottom(final int height)
{
    return repeatCharSequence("-", height * 2 + 1);
}
```

Wie schon bei anderen Aufgaben demonstriert, ist es oftmals vorteilhaft, Funktionalitäten in separate Hilfsmethoden auszulagern, hier für das Wiederholen eines Zeichens. In Java 11 nutzt man die Methode `repeat()`. Ansonsten schreibt man sich eine Hilfsmethode `repeatCharSequence()` (vgl. Abschnitt 2.3.7).

Alle Türme ausgeben Schließlich kombinieren wir die Ausgabefunktionalität in der folgenden Methode, um die drei als Listen repräsentierten Türme nebeneinander auszugeben:

```
static void printTowers(final int maxHeight,
                        final Tower source,
                        final Tower helper,
                        final Tower destination)
{
    final List<String> tower1 = source.printTower(maxHeight);
    final List<String> tower2 = helper.printTower(maxHeight);
    final List<String> tower3 = destination.printTower(maxHeight);

    for (int i = 0; i < tower1.size(); i++)
    {
        final String line = tower1.get(i) + "   " +
                            tower2.get(i) + "   " +
                            tower3.get(i);

        System.out.println(line);
    }
}
```

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
jshell> solveHanoi(2)
Tower Of Hanoi 2
   A         B         C
   |         |         |
  #|#        |         |
 ##|##       |         |
-------   -------   -------
Moving slice 1: Tower [A] -> Tower [B]
   A         B         C
   |         |         |
   |         |         |
 ##|##      #|#        |
-------   -------   -------
Moving slice 2: Tower [A] -> Tower [C]
   A         B         C
   |         |         |
   |         |         |
   |        #|#      ##|##
-------   -------   -------
Moving slice 1: Tower [B] -> Tower [C]
   A         B         C
   |         |         |
   |         |        #|#
   |         |       ##|##
-------   -------   -------
```

8.4.2 Lösung 2: Edit Distance (★★★★☆)

Berechnen Sie für zwei Strings, wie viele Änderungen diese – ohne Beachtung der Groß- und Kleinschreibung – auseinander liegen, also wie man den einen String in den anderen durch ein- oder mehrmaliges Anwenden einer beliebigen der folgenden Aktionen überführen kann:

- Ein Zeichen hinzufügen ($+$),
- ein Zeichen löschen ($-$) oder
- ein Zeichen verändern ($\rightsquigarrow$).

Schreiben Sie eine Methode `int editDistance(String, String)`, die zeichenweise die drei Aktionen testet und den anderen Teil rekursiv prüft.

Beispiele Für die gezeigten Eingaben sind folgende Modifikationen nötig:

Eingabe 1	Eingabe 2	Resultat	Aktionen
"Micha"	"Michael"	2	Micha $\underset{+\,e}{\rightarrow}$ Michae $\underset{+\,l}{\rightarrow}$ Michael
"Ananas"	"Banane"	3	Ananas $\underset{+\,B}{\rightarrow}$ BAnanas $\underset{-\,s}{\rightarrow}$ BAnana $\underset{a\,\rightsquigarrow\,e}{\rightarrow}$ BAnane

Algorithmus Beginnen wir zu überlegen, wie wir hier vorgehen können: Wenn beide Strings übereinstimmen, so ist die Edit Distance 0. Wenn einer der beiden Strings keine Zeichen (mehr) enthält, dann ist die Distanz zum anderen die Anzahl der Zeichen des anderen Strings – das würde mehrmaliges Einfügen der korrespondierenden Zeichen erfordern. Dies definiert den rekursiven Abbruch.

Ansonsten prüfen wir beide Strings von deren Anfang und vergleichen Zeichen für Zeichen. Sind diese gleich, so gehen wir eine Position weiter Richtung Ende des Strings. Bei Abweichung prüfen wir drei verschiedene Varianten:

1. Insert: rekursiver Aufruf für die nächsten Zeichen
2. Remove: rekursiver Aufruf für die nächsten Zeichen
3. Replace: rekursiver Aufruf für die nächsten Zeichen

```
static int editDistance(final String str1, final String str2)
{
    return editDistanceRec(str1.toLowerCase(), str2.toLowerCase());
}

static int editDistanceRec(final String str1, final String str2)
{
    // rekursiver Abbruch
    // beide stimmen überein
    if (str1.equals(str2))
       return 0;
```

```
    // wenn einer der Strings am Anfang ist und der andere
    // noch nicht, dann nimm die Länge des verbliebenen Strings
    if (str1.length() == 0)
       return str2.length();
    if (str2.length() == 0)
       return str1.length();

    // Prüfe, ob die Zeichen übereinstimmen, und dann auf zum nächsten
    if (str1.charAt(0) == str2.charAt(0))
    {
       // rekursiver Abstieg
       return editDistance(str1.substring(1), str2.substring(1));
    }
    else
    {
       // rekursiver Abstieg: Prüfe auf insert, delete, change
       final int insertInFirst = editDistanceRec(str1.substring(1), str2);
       final int deleteInFirst = editDistanceRec(str1, str2.substring(1));
       final int change = editDistanceRec(str1.substring(1), str2.substring(1));

       // Minimum aus allen drei Varianten + 1
       return 1 + minOf3(insertInFirst, deleteInFirst, change);
    }
}

static int minOf3(final int x, final int y, final int z)
{
    return Math.min(x, Math.min(y, z));
}
```

Die gezeigte Variante ist recht gut verständlich, was an sich schon ein Vorteil ist. Jedoch werden durch die Aufrufe von `substring()` ziemlich viele Teilstrings temporär erzeugt. Wie geht es besser?

Bevor wir uns Gedanken zu Optimierungen machen und diesbezüglich Änderungen im Sourcecode vornehmen, sollten wir zunächst einmal messen, ob das überhaupt notwendig ist. Zudem ist es sehr sinnvoll, Unit Tests zu erstellen, die zum einen initial prüfen, ob unsere Implementierung wie gewünscht arbeitet, und zum anderen bei den Optimierungen ein Sicherheitsnetz bilden und uns zeigen können, ob wir dadurch keine Fehler eingebaut haben.

Prüfung

Zum Nachvollziehen rufen wir die gerade erstellte Methode für einige Eingabewerte in einem Unit Test auf:

```
@ParameterizedTest(name = "edit distance between {0} and {1} is {2}")
@CsvSource({ "Micha, Michael, 2", "Ananas, Banane, 3" })
void editDistance(String input1, String input2, int expected)
{
    var result = Ex02_EditDistance.editDistance(input1, input2);

    assertEquals(expected, result);
}
```

Außerdem wollen wir einmal schauen, wie die Performance ist – weil es nur auf eine grobe Einordnung ankommt, ist hier die Genauigkeit von `currentTimeMillis(()` absolut ausreichend:

```
public static void main(final String args[])
{
    final String[][] inputs_tuples = { { "Micha", "Michael"},
                                       { "Ananas", "Banane" },
                                       { "sunday-Morning",
                                         "saturday-Night" },
                                       { "sunday-Morning-Breakfast",
                                         "saturday-Night-Party" } };

    for (final String[] inputs : inputs_tuples)
    {
        final long start = System.currentTimeMillis();
        System.out.println(inputs[0] + " -> " + inputs[1] +
                           " edits: " + editDistance(inputs[0], inputs[1]));
        final long end = System.currentTimeMillis();
        System.out.println("editDist took " + (end - start) + " ms");
    }
}
```

Führen wir die obigen Zeilen aus, so kommt es mit (viel) Geduld in etwa zu folgenden Ausgaben – tatsächlich habe ich die letzte Berechnung nach einigen Minuten abgebrochen, weshalb sie hier nicht gezeigt ist:

```
Micha -> Michael edits: 2
editDist took 0 ms
Ananas -> Banane edits: 3
editDist took 1 ms
sunday-Morning -> saturday-Night edits: 9
editDist took 6445 ms
```

Die Laufzeiten nehmen deutlich zu, je stärker sich die beiden Eingaben unterscheiden. Schon beim dritten Vergleich noch recht kurzer Strings haben wir rund 6 Sekunden Laufzeit.

Machen wir uns zweistufig an eine Verbesserung. Wir schauen uns zunächst an, wie man die vielen temporären Strings vermeidet und welchen Einfluss das besitzt. Schließlich bietet sich immer auch Memoization als Optimierung an. Wie das geht, zeigt uns die Lösung der Bonusaufgabe.

Optimierter Algorithmus Eine Optimierung zur Vermeidung der Erzeugung von vielen `String`-Objekten erzielt man, indem man Positionszeiger nutzt. In diesem Fall $pos1$ und $pos2$. Als kleine Abwandlung des Algorithmus starten die Vergleiche vom Ende der Strings. Somit vergleichen wir zeichenweise vom Ende und arbeiten uns in Richtung Anfang des Strings vor. Dabei gilt:

1. Insert: rekursiver Aufruf für die nächsten Zeichen, also $pos1$ und $pos2 - 1$
2. Remove: rekursiver Aufruf für die nächsten Zeichen, also $pos1 - 1$ und $pos2$
3. Replace: rekursiver Aufruf für die nächsten Zeichen, also $pos1 - 1$ und $pos2 - 1$

Dies führt zu folgender Implementierung:

```
static int editDistance(final String str1, final String str2)
{
    return editDistance(str1.toLowerCase(), str2.toLowerCase(),
                        str1.length() - 1, str2.length() - 1);
}

static int editDistance(final String str1, final String str2,
                        final int pos1, final int pos2)
{
    // rekursiver Abbruch
    // wenn einer der Strings am Anfang ist und der andere
    // noch nicht, dann nimm die Länge des verbliebenen Strings
    if (pos1 == 0)
        return pos2;

    if (pos2 == 0)
        return pos1;

    // Prüfe, ob die Zeichen übereinstimmen, und dann auf zum nächsten
    if (str1.charAt(pos1) == str2.charAt(pos2))
    {
        // rekursiver Abstieg
        return editDistance(str1, str2, pos1 - 1, pos2 - 1);
    }
    else
    {
        // prüfe auf insert, delete, change
        final int insertInFirst = editDistance(str1, str2, pos1, pos2 - 1);
        final int deleteInFirst = editDistance(str1, str2, pos1 - 1, pos2);
        final int change = editDistance(str1, str2, pos1 - 1, pos2 - 1);

        // Minimum aus allen drei Varianten + 1
        return 1 + minOf3(insertInFirst, deleteInFirst, change);
    }
}
```

Prüfung

Der Sourcecode ist etwas komplizierter geworden. Wiederum erstellen wir einen Unit Test, der exakt so aussieht wie zuvor und dessen Ausführung zeigt, dass die obige Implementierung weiterhin die erwarteten Resultate liefert.

Schauen wir uns nun an, ob sich eine Verbesserung in der Laufzeit ergibt, und nutzen dazu das initial gezeigte Programmgerüst. Das liefert folgende Laufzeiten:

```
Micha -> Michael edits: 2
editDist took 0 ms
Ananas -> Banane edits: 3
editDist took 0 ms
sunday-Morning -> saturday-Night edits: 9
editDist took 634 ms
```

Tatsächlich ist es für den dritten Fall deutlich schneller, jedoch habe ich auch hier nach einigen Minuten die Berechnung für das vierte Wertepaar abgebrochen.

Wir erkennen, dass Mikrooptimierungen zwar Verbesserungen mit sich bringen, jedoch keine signifikante Änderung bewirken. Das diskutierte ich recht ausführlich in

meinem Buch »Der Weg zum Java-Profi« [2]. Dort stelle ich auch dar, dass man Optimierungen auf höheren Ebenen, also Algorithmus, Design und Architektur, bevorzugen sollte. Hier bietet sich nun Memoization als Verbesserung im Algorithmus an.

Bonus: Optimieren Sie Edit Distance mit Memoization (★★★☆☆)

In der Einleitung hatte ich Memoization als Technik beschrieben und erwähnt, dass man oftmals eine Map als Zwischenspeicher nutzt. Weil wir das bereits kennengelernt haben, möchte ich nach einer kurzen Lösungsskizze, deren konkrete Implementierung Sie in den Sourcen des Begleitprojekts finden, eine Variante von Memoization zeigen.

Für die erste Variante kann man Memoization mit einer Map nutzen, jedoch benötigt man dann als Schlüssel einen Compound Key aus zwei Strings, was bis Java 14 zu einigem Sourcecode führt. Mit Java 14 nutzen wir einen Record:

```
record StringPair(String frist, String second)
{
}
```

Im Sourcecode sieht dies exemplarisch folgendermaßen aus:

```
// MEMOIZATION
final StringPair key = new StringPair(str1, str2);
if (memodata.containsKey(key))
    return memodata.get(key);
```

Wie wirkt sich das nun aus? Durch Einsatz von Memoization erhalten wir eine extreme Verbesserung bezüglich der Laufzeit:

```
Micha -> Michael edits: 2
editDist took 3 ms
Ananas -> Banane edits: 3
editDist took 6 ms
sunday-Morning -> saturday-Night edits: 9
editDist took 4 ms
sunday-Morning-Breakfast -> saturday-Night-Party edits: 16
editDist took 4 ms
```

Bitte bedenken Sie, dass `currentTimeMillis()` leicht ungenau ist, deswegen kann die Ausgabe bei einem Durchlauf 17 ms sein und bei einem anderen 5 ms. Wichtig ist die Größenordnung, die wir hier stark verbessert haben.

Variante für die zweite Implementierung Für die bereits optimierte Implementierung, die mit Positionen arbeitet, bietet sich zur Datenhaltung der Memoization eher ein `int[][]` an – beachten Sie bitte, dass wir das Array mit -1 vorinitialisieren. Ansonsten könnten wir keine Edit Distance von 0 für zwei Positionen erkennen.

```
static int editDistanceOptimized(final String str1, final String str2)
{
    final int length1 = str1.length();
    final int length2 = str2.length();
```

```
    var memodata = new int[length1][length2];
    for (int i = 0; i < length1; i++)
        for (int j = 0; j < length2; j++)
            memodata[i][j] = -1;

    return editDistanceWithMemo(str1.toLowerCase(), str2.toLowerCase(),
                                length1 - 1, length2 - 1, memodata);
}

static int editDistanceWithMemo(final String str1, final String str2,
                                final int pos1, final int pos2,
                                final int[][] values)
{
    // rekursiver Abbruch
    // wenn einer der Strings am Anfang ist und der andere
    // noch nicht, dann nimm die Länge des verbliebenen Strings
    if (pos1 == 0)
        return pos2;
    if (pos2 == 0)
        return pos1;

    // MEMOIZATION
    if (memodata[pos1][pos2] != -1)
        return memodata[pos1][pos2];

    int result = 0;
    // Prüfe, ob die Zeichen übereinstimmen, und dann auf zum nächsten
    if (str1.charAt(pos1) == str2.charAt(pos2))
    {
        // rekursiver Abstieg
        result = editDistanceWithMemo(str1, str2, pos1 - 1, pos2 - 1, values);
    }
    else
    {
        // prüfe auf insert, delete, change
        final int insertInFirst =
                  editDistanceWithMemo(str1, str2, pos1, pos2 - 1, values);
        final int deleteInFirst =
                  editDistanceWithMemo(str1, str2, pos1 - 1, pos2, values);
        final int change =
                  editDistanceWithMemo(str1, str2, pos1 - 1, pos2 - 1, values);
        // Minimum aus allen drei Varianten
        result = 1 + minOf3(insertInFirst, deleteInFirst, change);
    }
     // MEMOIZATION
    memodata[pos1][pos2] = result;

    return result;
}
```

Führen wir die gleichen Prüfungen wie zuvor aus, so ist diese nochmals leicht schneller als die erste Variante mit Memoization und selbst bei der letzten Berechnung der Edit Distance von 16 ist nur eine Laufzeit von einer Millisekunde ermittelbar:

```
Micha -> Michael edits: 2
editDist took 0 ms
Ananas -> Banane edits: 3
editDist took 0 ms
sunday-Morning -> saturday-Night edits: 9
editDist took 0 ms
sunday-Morning-Breakfast -> saturday-Night-Party edits: 16
editDist took 1 ms
```