

Der ganze Artikel basiert auf dem Buch von Cay Horstmann, *Big Java*, 4th Edition, Kapitel 13.

Korrekturen und Verbesserungsvorschläge bitte melden.



Table of Contents

Einleitung	2
Dreieckszahlen	2
Infinite Recursion - Stack Overflow	4
Rekursiv denken – Palindrome erkennen.....	5
Rekursive Helper Methoden	9
Die Effizienz von Rekursion (en)	9

Rekursion

Lerninhalt

- Technik der Rekursion lernen
- Den Unterschied zwischen Rekursion und Iteration verstehen
- „rekursiv denken“ lernen
- Verstehen, wann Rekursion die Effizienz eines Algorithmus beeinflusst



Einleitung

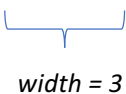
Rekursion ist eine Technik um komplexe Berechnungsproblemen zu vereinfachen. Das Wort „rekursiv“ bedeutet hier, dass die Selbe Berechnung wiederkehrt oder wiederholt wird, während das Problem gelöst wird. Oft ist rekursives Denken der natürliche Weg über ein Problem nachzudenken, und es gibt Problemstellungen, welche ohne Rekursion schwierig zu lösen sind. Wir werden einfache und komplexere rekursive Beispiele sehen und lernen „rekursiv zu denken“.

Dreieckszahlen

Wir betrachten ein einfaches Beispiel, welches die Anwendung von Rekursion zeigt. Wir starten mit einer Dreiecksform:

```
[ ]
[ ][ ]
[ ][ ][ ]
```

Fläche = 6



width = 3

Wir wollen die Fläche des Dreiecks berechnen mit der Breite n und der Annahme dass jedes [] Feld die Fläche 1 hat. Dieser Flächeninhalt wird auch als die n 'te Dreieckszahl bezeichnet. Am Dreieck oben, ist einfach ersichtlich, dass die Fläche 6 ist.

Sie wissen vielleicht, dass eine einfache Formel existiert um diese Zahlen zu berechnen. Das soll aber nicht das Ziel sein, sondern wir gehen davon aus, dass wir noch keine Ahnung über die Lösung haben. Das ultimative Ziel hier ist nicht die Zahl zu berechnen, sondern die rekursive Herangehensweise an das Problem.

Wir wollen folgende Klasse implementieren:

```
public class Triangle
{
    private int width;

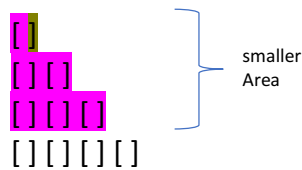
    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    public int getArea()
    {
        ...
    }
}
```

Ist die Breite des Dreiecks 1, dann ist die Fläche 1, weil es genau ein [] gibt.

```
public int getArea()
{
    if (width == 1) { return 1; }
    ...
}
```

Um den allgemeinen Fall zu behandeln, betrachten wir folgendes Bild:



Angenommen wir kennen die Fläche des kleineren, eingefärbten Dreiecks, dann berechnet sich die Fläche aus:

$$\text{smallerArea} + \text{width}$$

Wie können wir die eingefärbte kleinere Fläche berechnen? Dazu instanzieren wir ein Dreieck mit der kleineren Grösse und fragen nach der Fläche mit der Methode `getArea()`.

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Nun vervollständigen wir die `getArea` Methode.

```
public int getArea()
{
    if (width == 1) { return 1; }
    Triangle smallerTriangle = new Triangle(width - 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
```

Was passiert hier genau? Wir wollen das veranschaulichen:

```
int width = 4;
```

1. `getArea()` verkleinert das Dreieck auf `width = 3`
 - 1.1. vom verkleinerten Dreieck wird `getArea()` aufgerufen
 - 1.2. in diesem Aufruf wird ein neues Dreieck mit `width = 2` erstellt
 - 1.2.1. vom verkleinerten Dreieck wird `getArea()` aufgerufen
 - 1.2.2. dann wird das Dreieck wieder verkleinert auf `width = 1`
 - 1.2.2.1. `getArea()` gibt dann 1 zurück
 - 1.2.3. die Methode gibt `smallerArea + width = 1 + 2 = 3` zurück
 - 1.3. die Methode gibt `smallerArea + width = 3 + 3 = 6` zurück
2. die Methode gibt `smallerArea + width = 6 + 4 = 10` zurück.

Eine rekursive Berechnung löst ein Problem, in dem die Lösung des gleichen Problems auf einfachere Eingabewerte angewendet wird.

Um das Flächenproblem zu lösen, verwenden wir die Tatsache, dass das selbe Problem mit einer kleineren Breite gelöst werden kann. Das nennen wir dann eine *rekursive* Lösung.


Die verschachtelten Aufrufe einer rekursiven Methode wirken kompliziert und am besten versucht man gar nicht erst darüber zu studieren. Besser ist es, das Problem auf eine offensichtliche Art wie bei der `getArea` Methode anzugehen. Im einfachsten Fall ist die Breite gleich 1, dann ist die Fläche 1. Die Fläche des grösseren Dreiecks ist die Summe aus kleinerem Dreieck und Breite.

Zwei Anforderungen, welche erfüllt sein müssen, damit Rekursion funktioniert:

1. Jeder rekursive Aufruf muss die Berechnung vereinfachen.
2. Die einfachsten Fälle müssen separat und ohne rekursiven Aufruf behandelt werden.

Die `getArea` Methode ruft sich selber mit immer kleinerer Breite auf, bis die Breite 1 erreicht wird. Die Breite 1 ist speziell behandelt, somit gibt die `getArea` Methode immer einen gültigen Wert zurück.

Damit eine Rekursion terminiert, braucht es Spezialfälle für die einfachsten Werte.


FRAGE 	<ol style="list-style-type: none"> 1. Was passiert, wenn die Methode <code>getArea</code> mit einer Breite von -1 aufgerufen wird? 2. Was sollte die <code>getArea</code> Methode zurückgeben für Breiten kleiner oder gleich Null?
---	---

Selbstverständlich kann die Dreiecksfläche mit einer einfachen Schleife berechnet werden:

$$1 + 2 + 3 + \dots + width = \text{Fläche}$$

oder noch einfacher mit der Summenformel: $\frac{(1+width) \cdot width}{2}$


Die rekursive Lösung für dieses Beispiel dient als „warm-up“ um das Konzept der Rekursion kennen zu lernen. Im Anhang finden Sie den Source Code zu diesem Beispiel.

SELF CHECK 	Wie würden Sie das Programm modifizieren, um die Fläche eines Quadrates zu berechnen? <pre> [] [] [] [] </pre>
--	---

Infinite Recursion - Stack Overflow

Ein häufiger Programmierfehler ist eine Rekursion, ohne Abbruchbedingung. Weil der Computer Memory benötigt um Methodenaufrufe und Rücksprungadressen zu speichern, kann nicht unendlich oft eine Methode sich selber aufrufen, dies führt zu einem „stack overflow“. Dies passiert entweder wenn die Methodenaufrufe nicht einfacher werden oder

weil die spezielle Abbruchbedingung für den einfachsten Fall fehlt. Dies würde z.B. passieren, wenn `getArea` mit einer negativen Breite aufgerufen wird und so die Rekursion nicht abbricht.

CHALLENGE 	Benutzen Sie Ihre IDE und den Debugger um die Methode <code>getArea</code> anhand des Codes im Anhang zu debuggen. Setzen Sie den BP ¹ auf den Anfang der Methode und inspizieren sie die <code>width</code> Variable, welche den Wert 10 hat. Entfernen Sie den BP und gehen Sie weiter zur Zeile <code>return smallerArea + width;</code> Die Variable <code>width</code> hat nun den Wert 2. Was ist hier los, es gab dazwischen keine Instruktion den Wert von <code>width</code> zu verändern, was ist passiert? Tipp: Um den Überblick nicht zu verlieren, müssen Sie den Call-Stack im Debugger im Auge behalten.
---	--

Rekursiv denken – Palindrome erkennen

Ein Problem rekursiv zu lösen, erfordert eine andere Denkweise als das Problem in einer `for`- oder `while`-Schleife zu lösen. Es hilft, etwas faul zu sein, und sich mal vorerst um die einfachen Fälle zu kümmern. Dann müssen wir nur noch einen Weg finden, die einfachen Lösungen so zu verwenden, damit das ganze Problem gelöst werden kann.

Um die Technik zu verstehen, wollen wir folgendes Problem untersuchen. Wir wollen testen, ob ein Satz ein Palindrom ist oder nicht.

Palindrome gibt es in verschiedensten Sprachen hier ein paar typische Beispiele:

- Sugus!
- Madam, I'm Adam.
- A man, a plan, a canal, Panama!
- Go hang a salami, I'm a lasagna hog.
- Ein Neger mit...

Wenn wir Palindrome untersuchen, spielen Gross-Kleinschreibung, Leerzeichen, Satz- und Trennzeichen keine Rolle.

¹ Break Point

Wir wollen also in der Klasse Sentence die Methode isPalindrome implementieren.

```
public class Sentence
{
    private String text;

    /**
     * Constructs a sentence.
     * @param aText the stripped sentence
     */
    public Sentence(String aText)
    {
        text = aText;
    }

    /**
     * Tests for a palindrome.
     * @return true if is a palindrome
     */
    public boolean isPalindrome()
    {
        ...
    }
}
```

Schritt 1

Wir müssen uns überlegen, wie die Inputdaten so vereinfacht werden könne, dass das Problem auf den einfacheren Input angewendet werden kann. Vereinfachen heisst, wir wollen etwas vom originalen Input entfernen, z.B. ein oder zwei Zeichen von einem Text entfernen, einen Teil einer geometrischen Figur entfernen oder oft den Input in zwei Teile trennen und dann das Problem für beide Hälften zu lösen versuchen.

Für das Palindrom-Problem, sind folgend ein paar Ansätze aufgelistet:

- Entferne erstes Zeichen
- Entferne letztes Zeichen
- Entferne erstes und letztes Zeichen
- Split Text in zwei Hälften

Schritt 2

Kombiniere Lösungen mit einfacherem Input in eine Lösung des originalen Problems.

Wir betrachten also die Ansätze in Schritt 1. Hier sollte man sich nicht darum kümmern, wie diese Lösungen gefunden werden könnten, wir verlassen uns faul darauf, dass wir den Input vereinfacht haben und jemand anderes die Lösung finden wird.

Überlegen wir uns also, wie wir die Lösung für den vereinfachten Input so verwenden können, damit wir unser aktuelles Problem lösen können, an dem wir uns gerade die Zähne ausbeissen. Vielleicht muss man dazu zwei Lösungen für die beiden Hälften addieren oder sonst eine Überlegung anstellen.

Unser Palindrom zu halbieren, scheint keine gute Idee zu sein.

Sugus

wird zu

su + g + us oder

sug + us

und kein Teil ist wieder ein Palindrom, d.h. wir kommen mit Rekursion nicht weiter.

Am erfolgversprechendsten scheint das erste und letzte Zeichen zu entfernen.

S ugu s wird zu **ugu**.

Angenommen, wir können den kürzeren Text auf Palindrom prüfen, dann ist der original Text auch ein Palindrom, da wir den selben Buchstaben vorne und hinten entfernt haben, ein **s**. Das klingt vielversprechend, ein Text ist also ein Palindrom, wenn

1. der erste und letzte Buchstabe identisch sind und
2. der Text den wir erhalten, wenn erster und letzter Buchstabe entfernt wird, auch ein Palindrom ist

Also, wir kümmern uns nicht um die Frage, wie der Test funktioniert für den kürzeren Text **ugu**. Es funktioniert einfach.

Einen weiteren Fall müssen wir noch behandeln. Was wenn der erste oder letzte Buchstabe gar kein Buchstabe ist, sondern ein Sonderzeichen, z.B.

Madam, I 'm Adam.

endet mit einem Punkt und dieser ist nicht identisch mit dem ersten Buchstaben M. Das ist kein Problem, denn wir haben am Anfang definiert, dass wir Sonderzeichen nicht berücksichtigen wollen. Wir müssen also in diesem Fall, nur den Punkt am Ende entfernen, und nicht auch das M am Anfang. Dasselbe gilt natürlich auch für Nicht-Buchstaben am Anfang. Wir kommen somit zu folgenden kompletten Testfällen:

- Sind erstes und letztes Zeichen beides Buchstaben, prüfe ob sie gleich sind. Falls ja, entferne beide und prüfe den kürzeren Text.
- Sonst, falls das letzte Zeichen kein Buchstabe ist, entferne es, und teste den kürzeren Text.
- Sonst, falls das erste Zeichen kein Buchstabe ist, entferne es, und teste den kürzeren Text.

In allen drei Fällen, können wir die Lösung für den vereinfachten Text verwenden, um das originale Problem zu lösen.

Schritt 3

Finde Lösungen für die einfachsten Eingabetexte.

Eine rekursive Berechnung vereinfacht laufend den Input. Irgendwann erreicht die Rekursion sehr einfache Inputs. Um sicherzustellen, dass die

Rekursion stoppt, müssen diese einfachsten Eingaben separat behandelt werden. Dies ist in der Regel sehr einfach.

Unter Umständen landet man so bei unwirklichen Inputs, wie leere Strings oder Figuren ohne Fläche. In solchen Fällen studiert man am besten einen etwas grösseren Eingabewert und schaut, was für einen Wert denn zu einem solch unwirklichen Input zugeordnet werden sollte, so dass die Fälle im Schritt 2 korrekte Werte liefern.

Die einfachsten Eingaben für unseren Palindrom Test sind:

- Text mit zwei Zeichen
- Text mit einem Zeichen
- Leerer Text

Den Fall für Texte mit zwei Zeichen müssen wir nicht separat behandeln, denn gemäss unserem Vorgehen entfernen wir entweder einen oder beide Zeichen. Wir müssen uns aber um Texte mit ein oder keinem Zeichen kümmern, denn für diese haben wir kein Vorgehen zum Vereinfachen.

Der leere Text ist ein Palindrom. Sehen Sie dazu die Frage unten. Ein einzelner Buchstabe ist ein Palindrom. Was wenn das einzelne Zeichen kein Buchstabe ist, z.B. „!“ . Gemäss unserem Vorgehen führt dies zu einem leeren Text und der ist dann ein Palindrom, also Fall geklärt.

Wir halten fest: alle Texte mit einem oder keinem Zeichen sind Palindrome.

Schritt 4	<p>Implementieren der Lösung, indem die einfachsten Fälle und die Vereinfachung der Inputdaten kombiniert werden.</p> <p>Die einfachen Fälle, welche wir in Schritt 3 gefunden haben, müssen im Code abgehandelt werden. Für Inputdaten, welche nicht den einfachsten Fällen entsprechen, muss die Logik aus Schritt 2 implementiert werden.</p>
-----------	--

FRAGE



Warum, denken Sie, ist der leere Text ein Palindrom?

Hier folgt die Palindrom Methode:

```
public boolean isPalindrome()
{
    int length = text.length();

    // Separate case for shortest strings.
    if (length <= 1) { return true; }

    // Get first and last characters, converted to lowercase.
    char first = Character.toLowerCase(text.charAt(0));
    char last = Character.toLowerCase(text.charAt(length - 1));

    if (Character.isLetter(first) && Character.isLetter(last))
    {
        // Both are letters.
    }
}
```



```

        if (first == last)
        {
            // Remove both first and last character.
            Sentence shorter = new Sentence(text.substring(1, length - 1));
            return shorter.isPalindrome();
        }
        else
        {
            return false;
        }
    }
    else if (!Character.isLetter(last))
    {
        // Remove last character.
        Sentence shorter = new Sentence(text.substring(0, length - 1));
        return shorter.isPalindrome();
    }
    else
    {
        // Remove first character.
        Sentence shorter = new Sentence(text.substring(1));
        return shorter.isPalindrome();
    }
}

```

FRAGE

Wann stoppt die rekursive isPalindrome Methode sich selber aufzurufen?

Rekursive Helper Methoden

In manchen Fällen kann eine rekursive Lösung einfacher gefunden werden, wenn das originale Problem leicht abgeändert wird.

Im Beispiel vom Palindromtest, ist es ineffizient, in jedem Schritt ein neues Sentence Objekt zu erstellen, welches dann den ganzen String testet. Zum Vereinfachen können wir eine Helper-Methode erstellen

```

/**
 * Testet einen Substring auf Palindrom.
 * @param start Index des ersten Zeichens des Substrings
 * @param stop Index des letzten Zeichens des Substrings
 * @return true falls Substring ein Palindrom ist
 */
public boolean isPalindrome(int start, int end)

```

welche einen Substring testet. In den rekursiven Aufrufen werden dann lediglich die start und stop Werte angepasst, um passende Zeichenpaare zu überspringen. Es ist somit nicht mehr nötig, neue Sentence Objekte zu erstellen um den verkürzten String zu representieren. Für den interessierten Leser sei hier auf das entsprechende Kapitel im Buch verwiesen.

Die Effizienz von Rekursion (en)

As you have seen in this chapter, recursion can be a powerful tool to implement complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

Consider the Fibonacci sequence: a sequence of numbers defined by the equation

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is $34 + 55 = 89$.

We would like to write a function that computes f_n for any value of n . Let us translate the definition directly into a recursive method:

ch13/fib/RecursiveFib.java

```

1  import java.util.Scanner;
2
3  /**
4      This program computes Fibonacci numbers using a
5      recursive method.
6  */
7  public class RecursiveFib
8  {
9      public static void main(String[] args)
10     {
11         Scanner in = new Scanner(System.in);
12         System.out.print("Enter n: ");
13         int n = in.nextInt();
14
15         for (int i = 1; i <= n; i++)
16         {
17             long f = fib(i);
18             System.out.println("fib(" + i + ") = " + f);
19         }
20
21         /**
22             Computes a Fibonacci number.
23             @param n an integer
24             @return the nth Fibonacci number
25         */
26         public static long fib(int n)
27         {
28             if (n <= 2) { return 1; }
29             else return fib(n - 1) + fib(n - 2);
30         }
31     }

```

Program Run

Enter n: 50

fib(1) = 1

fib(2) = 1

fib(3) = 2

fib(4) = 3

```
fib(5) = 5  
fib(6) = 8  
fib(7) = 13
```

```
...
```

```
fib(50) = 12586269025
```

That is certainly simple, and the method will work correctly. But watch the output closely as you run the test program. The first few calls to the `fib` method are fast. For larger values, though, the program pauses an amazingly long time between outputs.

That makes no sense...