Inhaltsverzeichnis

Rekursive Unterprogramme

1. Das Beispiel fakultaet

In Java (und in vielen anderen modernen Programmiersprachen) darf ein Unterprogramm UP nicht nur *andere* Unteprogramme UP1, UP2, ... aufrufen, sondern auch *sich selbst*. Ein Unterprogramm, welches so "auf sich selbst zurückgreift" bezeichnet man als ein *rekursives Unterprogramm*.

Unter der *Fakultät einer natürlichen Zahl* `n` versteht man das Produkt aller natürlichen Zahlen von `1` bis `n`. Es ist sinnvoll festzulegen, dass die *Fakultät von* `0` gleich `1` ist (und nicht etwa gleich `0`). Hier ein *rekursives* Unterprogramm (aus dem Beispielprogramm `RekFakultaet01`) zum Berechnen der Fakultät einer natürlichen Zahl *n*:

```
 1    static public long fakultaet(long n) {
 2       // Liefert die Fakultaet von n. Falls n unzulaessig (d.h. negativ)
 3       // ist, wird 1 als Ergebnis geliefert.
 4
 5       if (n <= 1) {
 6          return 1;                 // Einfacher (nicht-rekursiver) Fall
 7       } else {
 8          return n * fakultaet(n-1); // Rekursiver Fall
 9       }
10    } // fakultaet
```

Aufgabe: Schreiben Sie ein Programm namens `RekFakultaet02`, mit dem man die Fakultaet von natürlichen Zahlen bis 100 berechnen kann. Benutzen Sie dazu die Klasse `java.math.BigInteger`.

Würde ein Unterprogramm UP sich in *jedem* Fall rekursiv aufrufen (unabhängig davon, mit welchen Parametern man es aufruft), dann käme die Rekursion *nie zu einem Ende*. Eine solche *Endlosrekursion* hat Ähnlichkeit mit einer *Endlosschleife*.

Weil man eine Endlosrekursion (in aller Regel) vermeiden will, muss der *Rumpf* eines rekursiven Unterprogramms im Wesentlichen aus einem *Fallunterscheidungs-Befehl* bestehen, in Java also aus einer `if`- oder einer `switch`-Anweisung. Dieser Fallunterscheidungs-Befehl muss zwei Arten von Fällen unterscheiden: *Einfache Fälle*, in denen das Unterprogramm sich *nicht* rekursiv aufruft und *rekursive Fällen*, in denen das Unterprogramm sich selbst aufruft.

Der Rumpf der obigen Funktion `fakultaet` besteht aus einer `if`-Anweisung, die *einen* einfachen Fall und *einen* rekursiven Fall unterscheidet. Im allgemeinen kann es mehrere einfache und mehrere rekursive Fälle geben.

2. Rekursive Funktionen mit Papier und Bleistift ausführen

Wenn man *rekursive Unterprogramme* "von Hand" ausführt besteht die Gefahr, dass man sich beim Ausführen der rekursiven Aufrufe verhaspelt und bald nicht mehr weiter weiss. Beim Ausführen von rekursiven *Funktionen* kann man das vermeiden, indem man eine *Wertetabelle* verwendet. In

die trägt man alle schon berechneten Funktionsergebnisse (mit den zugehörigen aktuellen Parametern) ein. Kommt man zu einem rekursiven Aufruf der Funktion, so führt man ihn *nicht* aus, sondern holt sich sein Ergebnis aus der Wertetabelle.

Aufgabe: Führen Sie Funktion *fakultaet* mit verschiedenen aktuellen Parametern (0, 1, 2 ...) und mit Hilfe der folgenden Wertetabelle aus:

n	0	1	2	3	4	5	...
fakultaet(n)							...

3. Das Beispiel fibo mit den Robotern die Roboter bauen

Die Fakultät einer natürlichen Zahl könnte man statt mit *Rekursion* mindestens genauso gut mit einer *Schleife* berechnen. Es gibt aber eine Reihe von Problemen, bei denen eine *rekursive* Lösung *einfacher* und "*natürlicher*" ist als eine Lösung mit *Schleifen*. Hier ein Beispiel (aus dem Programm `RekRoboter`):

```
 1    public static long fibo(long anzahlEltern, long anzahlTage) {
 2       // Wenn anzahlEltern viele Roboter in der Lage sind, pro Tag einen
 3       // weiteren Roboter zu bauen, und wenn man zu Beginn des ersten Pro-
 4       // duktionstages anzahlEltern viele Roboter hat, wieviele Roboter hat
 5       // man dann nach anzahlTage vielen Produktionstagen? Die Funktion
 6       // fibo liefert die Antwort auf diese Frage.
 7
 8       if (anzahlTage == 0) {
 9          return anzahlEltern;
10       } else {
11          // Jetzt ruft sich die Funktion fibo rekursiv auf:
12          long   anzahlRoboterGestern = fibo(anzahlEltern, anzahlTage-1);
13          return anzahlRoboterGestern + anzahlRoboterGestern / anzahlEltern;
14       }
15    } // fibo
```

Anmerkung: Die Grundidee für diese Funktion wurde schon im Jahr 1228 von *Leonardo von Pisa* (auch *Fibonacci* genannt) in seinem Buch *Liber Abaci* in Pisa veröffentlicht. Die Folge `1, 1, 2, 3, 5, 8, 13, ...` der Fibonacci-Zahlen läßt sich *mit Rekursion* so beschreiben: Sei
$z_1 = 1$,
$z_2 = 1$ und
$z_{n+2} = z_{n+1} + z_n$.

Erst im 18. Jahrhundert gelang es, die Folge der Fibonacci-Zahlen *ohne Rekursion* zu definieren, durch die Formel:

$$z_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Viele Menschen finden die die *rekursive* Beschreibung der Folge *einfacher* und *verständlicher* als die *nicht-rekursive* Beschreibung. Das gilt auch für die Lösungen von zahlreichen anderen Problemen.

4. Die Parameter müssen immer kleiner werden. Manchmal ist 15 kleiner als 14!

Das Beispielprogramm `RekKleiner` illustriert anhand einer rekursiven Funktion `rekFunk01` mit nur einem Parameter vom Typ `int` eine wichtige, allgemein gültige Tatsache. Die Funktion `rekFunk02` leistet genau *das Gleiche* wie `rekFunk01`, ist aber ein bisschen *kompakter notiert*.

```
1   // Datei RekKleiner.java
2   /* ------------------------------------------------------------------------
3   Wenn ein rekursives Unterprogramm mit einem Parameter n1 aufgerufen wird
4   und sich dann mit einem Parameter n2 rekursiv aufruft, dann muss n2 "in
5   irgendeinem Sinne kleiner sein als n1".
6
7   Fuer die rekursiven Funktionen in diesem Beispiel gilt, dass z.B. 15
8   kleiner ist als 14, 13, 12 und 11, aber *nicht* kleiner ist als 10.
9   Hier ist 15 auch nicht groesser als 10, sondern man sagt: 15 und 10
10  sind unvergleichbar.
11  ------------------------------------------------------------------------ */
12  class RekKleiner {
13     static final String BLANK = " ";
14     // ---------------------------------------------------------------------
15     static String rekFunk01(int n) {
16        // Liefert einen String mit ein bis fuenf aufeinanderfolgenden
17        // Ganzzahlen darin. Die erste Zahl ist immer gleich n, die letzte
18        // der Zahlen ist durch 5 teilbar.
19
20        if (n % 5 == 0) {
21           return "" + n; // Hier ist '+' die Konkatenation, nicht Addition!
22        } else {
23           return n + BLANK + rekFunk01(n+1);
24        }
25     } // rekFunk01
26     // ---------------------------------------------------------------------
27     static String rekFunk02(int n) {
28        // Leistet genau das Gleiche wie rekFunk01 (siehe oben), benutzt aber
29        // den dreistelligen "if-Operator" BEDINGUNG ? AUSDRUCK1 : AUSDRUCK2
30        return (n % 5 == 0) ? ("" + n) : (n + BLANK + rekFunk02(n+1));
31     } // rekFunk01
32     // ---------------------------------------------------------------------
33     ...
34  } // class RekKleiner
```

Aufgabe: Führen Sie Funktion `rekFunk01` mit verschiedenen aktuellen Parametern und mit Hilfe der folgenden *Wertetabelle* aus:

n	...	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	...
rekfunk01(n)	...																...

5. Eine schwierige Rekursion und wie man sie beschleunigen kann

Ein komplizierteres Beispiel für Rekursion ist das Programm `RekPlusbaeume01`. Es berechnet, auf wieviele Weisen man eine natürliche Zahl als *Summe* darstellen kann.

```
1   // Datei RekPlusbaeume01.java
2   /* ------------------------------------------------------------------------
3   Ein Plusbaum ist ein voll geklammerter Ausdruck, der aus Ganzzahl-Literalen,
4   Pluszeichen und Klammern besteht, z.B. ((2+2)+(5+(3+2))). Als Grenzfall
5   gilt auch eine Ganzzahl allein als Plusbaum, z.B. 7.
6
7   Es gibt genau 2 Plusbaeume mit dem Wert 2, naemlich 2 und (1+1).
8
9   Es gibt genau 5 Plusbaeume mit dem Wert 3, naemlich:
10  3, (1+2), (1+(1+1)), (2+1) und ((1+1)+1).
11
```



```
12  Gegeben eine positive Ganzzahl n, wieviele Plusbaeume mit dem Wert n gibt
13  es dann? Die Funktion anzPlusbaeume liefert die Antwort.
14  ------------------------------------------------------------------------ */
15  class RekPlusbaeume01 {
16     // ---------------------------------------------------------------------
17     static long anzPlusbaeume(int n) {
18        // Liefert die Anzahl der Plusbaeume die den Wert n haben. Dabei
19        // sollte n positiv (groesser gleich 1) sein.
20        if (n <= 1) return 1;
21
22        long erg = 1; // Die unzerlegte Zahl n ist *ein* Plusbaum.
23
24        // Alle Zerlegungen von n in 2 Summanden werden erzeugt:
25        for (int summand1=1; summand1<n; summand1++) {
26           int summand2 = n - summand1;
27           // Jetzt gilt: summand1 + summand2 = n
28
29           // Jeder Plusbaum fuer summand1 kann mit
30           // jedem Plusbaum fuer summand2 kombiniert werden:
31           erg += anzPlusbaeume(summand1) * anzPlusbaeume(summand2);
32        }
33        return erg;
34     } // RekPlusbaeume01
35     // ---------------------------------------------------------------------
36     static public void main(String[] _) {
37        // Fuer einige Ganzzahlen n wird die Anzahl der Plusbaeume mit Wert
38        // n berechnet und ausgegeben:
39
40        for (int n=1; n<=18; n++) {
41           long anzahl = anzPlusbaeume(n);
42           AM.p(n,       3); AM.p  (" hat ");
43           AM.p(anzahl, 20); AM.pln(" Plusbaeume.");
44        }
45
46        AM.pln("RekPlusbaeume01: Das war's erstmal!");
47     } // main
48     // ---------------------------------------------------------------------
49  } // RekPlusbaeume01
50  /* ------------------------------------------------------------------------
51  Ausgabe des Programms RekPlusbaeume:
52
53    1 hat                   1 Plusbaeume.
54    2 hat                   2 Plusbaeume.
55    3 hat                   5 Plusbaeume.
56    4 hat                  15 Plusbaeume.
57    5 hat                  51 Plusbaeume.
58    6 hat                 188 Plusbaeume.
59    7 hat                 731 Plusbaeume.
60    8 hat                2950 Plusbaeume.
61    9 hat               12235 Plusbaeume.
62   10 hat               51822 Plusbaeume.
63   11 hat              223191 Plusbaeume.
64   12 hat              974427 Plusbaeume.
65   13 hat             4302645 Plusbaeume.
66   14 hat            19181100 Plusbaeume.
67   15 hat            86211885 Plusbaeume.
68   16 hat           390248055 Plusbaeume.
69   17 hat          1777495635 Plusbaeume.
70   18 hat          8140539950 Plusbaeume.
71  RekPlusbaeume01: Das war's erstmal!
72  ------------------------------------------------------------------------ */
```

Beim Ausführen des Programms `RekPlusbaeume01` auf einem PC, Baujahr 2000, treten vor Ausgabe der letzten Zeilen spürbare *Verzögerungen* auf. Offensichtlich hat der Prozessor Mühe zu berechnen, dass es für die Zahl 18 mehr als 8 Milliarden Plusbäume gibt.

Das Beispielprogramm `RekPlusbaeume02` zeigt, wie man eine rekursive Funktion *beschleunigen* kann, indem man die bereits berechneten Ergebnisse z.B. in einer *Reihung* speichert (statt sie jedesmal erneut zu berechnen, wenn man sie noch mal braucht).

```
1     // ---------------------------------------------------------------------
2     static long anzPlusbaeume(int n) {
3        // Liefert die Anzahl der Plusbaeume die den Wert n haben. Dabei
4        // sollte n positiv (groesser gleich 1) sein.
5        // Zur Beschleunigung wird eine Reihung (namens tab) verwendet.
6
7        long[] tab = new long[n+1]; // tab[0] wird nicht benutzt
8
9        tab[1] = 1;
10       for (int i=2; i<tab.length; i++) {
11          tab[i] = 1;
12          for (int summand1=1; summand1<i; summand1++) {
13             int summand2 = i - summand1;
14             // Jetzt gilt: summand1 + summand2 = i
15             tab[i] += tab[summand1] * tab[summand2];
16          }
17       }
18
19       return tab[n];
20    } // anzPlusbaeume
21    // ---------------------------------------------------------------------
```

6. Rekursion und Iteration sind gleich mächtig

Programme, in denen *Schleifen* benutzt werden (statt Rekursion) bezeichnet man auch als *iterative* Programme. Grundsätzlich gilt: Jedes *iterativ* lösbare Problem ist auch *rekursiv* lösbar und umgekehrt, jedes *rekursiv* lösbare Problem ist auch *iterativ* lösbar. Bevor man diese Behauptung beweisen kann, muss man ihre Voraussetzungen präzisieren (was genau ist mit „einer iterativen Lösung“ bzw. „einer rekursiven Lösunge“ gemeint etc.).