

Inhaltsverzeichnis

| | |
|--|---|
| Rekursive Unterprogramme..... | 1 |
| 1. Das Beispiel fakultaet..... | 1 |
| 2. Rekursive Funktionen mit Papier und Bleistift ausführen..... | 1 |
| 3. Das Beispiel fibo mit den Robotern die Roboter bauen..... | 2 |
| 4. Die Parameter müssen immer kleiner werden. Manchmal ist 15 kleiner als 14!..... | 2 |
| 5. Eine schwierige Rekursion und wie man sie beschleunigen kann..... | 3 |
| 6. Bäume mit rekursiven Methoden bearbeiten..... | 5 |
| 7. Rekursion und Iteration sind gleich mächtig..... | 7 |

Rekursive Unterprogramme

1. Das Beispiel fakultaet

In Java (und in vielen anderen modernen Programmiersprachen) darf ein Unterprogramm UP nicht nur *andere* Unterprogramme 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 Fakultät von n. Falls n unzulässig (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 Fakultät 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

$$\begin{aligned}
 z_1 &= 1, \\
 z_2 &= 1 \text{ und} \\
 z_{n+2} &= z_{n+1} + z_n.
 \end{aligned}$$

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 funef 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 Konkatination, 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 positiven Ganz-
4 zahlen, Pluszeichen und Klammern besteht, z.B. ((2+2)+(5+(3+2))). Als
5 Grenzfall 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, wie viele 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         printf("%2d hat %,14d Plusbaeume.%n", n, anzahl);
43     }
44
45     printf("-----%n");
46     printf("RekPlusbaeume01: Das war's erstmal!%n");
47 } // main
48 // -----
49 // Eine Methode mit einem kurzen Namen:
50 static void printf(String f, Object... v) {System.out.printf(f, v);}
51 // -----
52 } // RekPlusbaeume01
53 /* -----
54 Ausgabe des Programms RekPlusbaeume01:
55
56 1 hat          1 Plusbaeume.
57 2 hat          2 Plusbaeume.
58 3 hat          5 Plusbaeume.
59 4 hat         15 Plusbaeume.
60 5 hat         51 Plusbaeume.
61 6 hat        188 Plusbaeume.
62 7 hat        731 Plusbaeume.
63 8 hat       2.950 Plusbaeume.
64 9 hat       12.235 Plusbaeume.
65 10 hat      51.822 Plusbaeume.
66 11 hat     223.191 Plusbaeume.
67 12 hat     974.427 Plusbaeume.
68 13 hat    4.302.645 Plusbaeume.
69 14 hat   19.181.100 Plusbaeume.
70 15 hat   86.211.885 Plusbaeume.
71 16 hat  390.248.055 Plusbaeume.
72 17 hat 1.777.495.635 Plusbaeume.
73 18 hat 8.140.539.950 Plusbaeume.
74 -----
75 RekPlusbaeume01: Das war's erstmal!
76 ----- */

```

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. Bäume mit rekursiven Methoden bearbeiten

Was ein binärer Baum ist, läßt sich besonders gut mit einer *rekursiven Definition* beschreiben:

Def.: Ein binärer Baum ist entweder ein *Leerbaum* oder ein *Knoten*, an dem zwei *Bäume* hängen (der linke und der rechte Unterbaum des Knotens).

Solche Bäume werden häufig als Sammlungen verwendet und müssen dann sortiert sein:

Def.: Ein binärer Baum ist sortiert, wenn jeder Knoten K einen Schlüssel hat und für jeden Knoten K gilt:

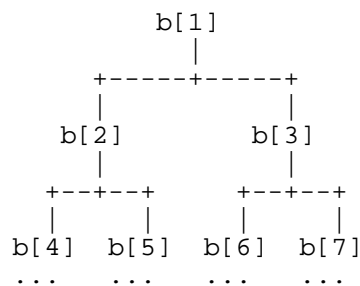
Der Schlüssel von K ist *größer* als alle Schlüssel in seinem *linken* Unterbaum und der Schlüssel von K ist *kleiner* als alle Schlüssel in seinem *rechten* Unterbaum.

Solche sortierten Bäume können besonders elegant mit *rekursiven Methoden* bearbeitet werden.

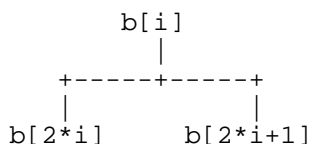
Einen Baum kann man unter anderem als eine *Reihung* b von Knoten implementieren. Der Knoten $b[0]$ bleibt (der Einfachheit halber) unbenutzt, und ansonsten gilt:

- $b[1]$ ist der Wurzelknoten des Baumes
- der *linke* Unterbaum eines Knotens $b[i]$ beginnt bei $b[2*i]$
- der *rechte* Unterbaum eines Knotens $b[i]$ beginnt bei $b[2*i+1]$,

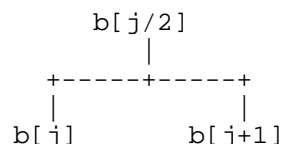
etwa so:



Allgemein gilt für jeden Index i



bzw. für jeden *geraden* Index j :



Durch einfache Indexrechnungen ($j/2$ oder $2*i$ oder $2*i+1$) kann man also in dem Baum "herumklettern", d.h. von einem Knoten zu seinem Vorgänger (Mutterknoten) oder zu seinen beiden Nachfolgern (Unterbäumen) gelangen.

Jedes Objekt der folgenden Klasse `UebBaum03` ist eine Sammlung, in der man `long`-Werte sammeln (einfügen und suchen) kann. Intern ist die Sammlung als ein binärer Baum strukturiert und dieser Baum ist als eine Reihung namens `baum` vom Typ `long[]` realisiert:

```
69 public class UebBaum03 {
70     // -----
71     private final long[] baum;    // Der Baum
72
73     // Noch nicht belegte Komponenten werden mit dem Wert LEER markiert:
74     private final long LEER = Long.MIN_VALUE;
75     // -----
76     public UebBaum03(int anzKnoten) {
77         this.baum = new long[anzKnoten];
78         for (int i=1; i<anzKnoten; i++) baum[i] = LEER;
79     } // Konstruktor UebBaum03
80     // -----
81     public boolean fuegeEin(long n) {
82         // Versucht, n in den baum einzufuegen, Liefert true wenn das
83         // Einfuegen gelingt oder n schon im baum enthalten war.
84         // Liefert false, wenn im baum fuer n kein Platz mehr ist.
85         return fuegeEinR(n, 1);
86     }
87
88     private boolean fuegeEinR(long n, int hier) {
89         // Wenn im baum kein Platz mehr fuer n ist:
90         if (hier >= baum.length) return false;
91
92         // Wenn n an der Stelle baum[hier] eingefuegt werden kann:
93         if (baum[hier] == LEER) {
94             baum[hier] = n; // n wird eingefuegt
95             return true;
96         }
97
98         // Wenn n im linken bzw. im rechten Unterbaum von baum[hier]
99         // eingefuegt werden muss:
100        if (n < baum[hier]) return fuegeEinR(n, 2*hier);
101        if (n > baum[hier]) return fuegeEinR(n, 2*hier+1);
102
103        // Jetzt gilt:
104        // n == baum[hier], d.h. n ist schon im baum:
105        return true;
106    } // fuegeEinR
107    // -----
108    public boolean istDrin(long n) {
109        // Liefert true genau dann wenn n im baum enthalten ist.
110        return istDrinR(n, 1);
111    } // istDrin
112
113    private boolean istDrinR(long n, int hier) {
114        if (hier >= baum.length) return false; // Ende des baum's erreicht
115        if (LEER == baum[hier]) return false; // Leeren baum-Teil erreicht
116
117        // Wenn n im linken bzw. rechten Unterbaum von baum[hier]
118        // gesucht werden muss:
119        if (n < baum[hier]) return istDrinR(n, 2*hier);
120        if (n > baum[hier]) return istDrinR(n, 2*hier+1);
121
122        // Jetzt gilt:
123        // n == baum[hier], d.h. wir haben n gefunden:
124        return true;
125    } // istDrinR
126    // -----
127    public void gibAus() {
128        // Gibt alle im baum enthaltenen Zahlen aufsteigend sortiert aus:
129        printf("Die Komponenten eines Baumes, aufsteigend sortiert:%n");
130        gibAusR(1);
131    } // gibAus
132
133    private void gibAusR(int hier) {
134        if (hier >= baum.length) return; // Ende des baum's erreicht
```

```
135     if (baum[hier] == LEER) return; // Leeren Teil des baum's erreicht
136
137     gibAusR(2*hier);           // Linken Unterbaum von baum[hier] ausgeben
138     printf("%4d ", baum[hier]); // Aktuellen Knoten baum[hier] ausgeben
139     gibAusR(2*hier+1);       // Rechten Unterbaum von baum[hier] ausgeben
140 } // gibAusR
141 // -----
142 // Eine Methode mit einem kurzen Namen:
143 static void printf(String f, Object... v) {System.out.printf(f, v);}
144 // -----
145 } // class UebBaum03
```

Die öffentliche Methode `fuegeEin` ist *nicht-rekursiv*. Sie ruft eine *rekursive* private Methode namens `fuegeEinR` auf, die "die eigentliche Arbeit macht". Die rekursive Methode bekommt den Parameter der nicht-rekursiven Methode und zusätzlich den Index 1 des Wurzelknotens.

Ganz Entsprechendes gilt auch für die Methoden `istDrin`/`istDrinR` und `gibAus`/`gibAusR`.

Es folgen ein paar Fragen zu den rekursiven Methoden:

Frage 1: Wie viele *nicht-rekursive* und wie viele *rekursive Fälle* werden im Rumpf der Methode `fuegeEinR` unterschieden?

Frage 2: Ebenso für die Methode `istDrinR`.

Frage 3: Ebenso für die Methode `gibAusR`.

Frage 4: Wie müsste man die Methode `gibAusR` ändern, damit sie die Komponenten der Sammlung *absteigend* sortiert ausgibt (statt *aufsteigend*)?

Antworten zu diesen Fragen findet man am Ende dieses Papiers.

7. Rekursion und Iteration sind gleich mächtig

Programme, in denen *Schleifen* benutzt werden (statt Rekursion) bezeichnet man auch als *iterative* Programme. Grundsätzlich gilt: Jedes *iterativ* lösbares Problem ist auch *rekursiv* lösbar und umgekehrt, jedes *rekursiv* lösbares 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 Lösung“ gemeint etc.).

Antworten zu den Fragen zur Klasse UebBaum03:

Antwort 1: Drei nicht-rekursive Fälle und zwei rekursive Fälle.

Antwort 2: Drei nicht-rekursive Fälle und zwei rekursive Fälle.

Antwort 3: Zwei nicht-rekursive Fälle und ein rekursiver Fall.

Antwort 4: Man braucht bloss die Reihenfolge der drei Befehle im rekursiven Fall umdrehen, etwa so:

```
69     gibAusR(2*hier+1);    // Rechten Unterbaum von baum[hier] ausgeben
70     printf("%4d ", baum[hier]); // Aktuellen Knoten baum[hier] ausgeben
71     gibAusR(2*hier);    // Linken Unterbaum von baum[hier] ausgeben
```