JAVA im Anfangsunterricht

Rekursiv definierte Funktionen

Alfred Nussbaumer

Rekursiv verwendete Funktionen lassen interessante und zum Teil schwierige Aufgabenstellungen zu. Trotz der einfachen Ausgabe auf der Konsole können dabei überraschende und unerwartete Ergebnisse studiert werden. Einige prominente Beispiele davon sollen in diesem Beitrag behandelt werden. Ihre Quellcodes stehen unter [8] zum Download bereit.

1. Grundlagen

Von einer Rekursion sprechen wir, wenn sich eine Methode selbst aufruft. Wird eine Abbruchbedingung nach einer endlichen Anzahl solcher „Selbstaufrufe“ erfüllt, so wird der aktuelle Methodenaufruf beendet, anschließend der vorletzte, dann der vorvorletzte ... usw. bis auch die erste Methode beendet wurde. Die Methode `arith()` ruft sich in der letzten Zeile selbst wieder auf („Endrekursion“) - beachten Sie bitte die Bedeutung der Variablen `start` für den Startwert und `dek` für das Dekrement, das bei jedem Aufruf der Methode vom aktuellen Startwert subtrahiert wird.

```
public class Rekursion {

  static void arith(int start, int dek) {
    if (start<0) return;
    System.out.println(start);
    arith(start-dek, dek);
  }

  public static void main (String args[]) {
    if (args.length==2) {
      int maxzahl = Integer.parseInt(args[0]);
      int dekrement = Integer.parseInt(args[1]);
      arith(maxzahl, dekrement);
    }
    else {
      System.out.println("Aufruf: java Rekursion
[zahl] [dekrement]");
    }
  }
}
```

Mit den folgenden Parametern erhalten wir eine arithmetische Folge:

```
nus@ice:~/java_bsp> java Rekursion 11 3
11
8
5
2
```

Will man die Folgenglieder in aufsteigender Reihenfolge, braucht man nur die Methode `arith()` abändern:

```
    static void arith(int start, int dek) {
        if (start<0) return;
        arith(start-dek, dek);
        System.out.println(start);
    }
```

Damit erhalten wir wunschgemäß:

```
nus@ice:~/java_bsp java Rekursionup 11 3
2
5
8
11
```

Was ist der Unterschied zwischen den beiden Versionen von `arith()`?

Natürlich können die Glieder einer arithmetischen Folge leicht iterativ mit Hilfe einer Zählschleife oder mit Hilfe einer Formel für das n-te Folgenglied berechnet werden. Welche Methode effizienter abläuft, hängt im Allgemeinen von der verwendeten Rechenoperationen ab.

2. Die Fakultäten-Funktion

Die Berechnung von $1! = 1$, $2! = 1 \cdot 2$, $3! = 1 \cdot 2 \cdot 3$, ... $n! = (n-1)! \cdot n$ ist ein Paradebeispiel einer rekursiv definierten Funktion:

```
public class fak {

  static int f(int n) {
    if (n==1) return 1;
    else return f(n-1)*n;
  }

  public static void main (String args[]) {
    if (args.length==1) {
      int n = Integer.parseInt(args[0]);
      System.out.println(n + "! = " + f(n));
    }
    else {
      System.out.println("Aufruf: java Rekursion
[zahl]");
       }
    }
}
```

Mit diesem Beispiel können nur kleine Fakultäten berechnet werden, weil Variablen vom Typ `int` verwendet wurden.

3. Die Fibonacci-Folge

Nach dem italienischen Mathematiker Leonardo Fibonacci ist folgende Funktion überliefert:

$a_1 = 1$

$a_2 = 1$

$a_n = a_{n-1} + a_{n-2}$ $(n>2)$.

Wir berechnen alle Folgenglieder rekursiv:

```
public class Fibonacci {

  static int fibonacci(int n) {
    if ((n==1) || (n==2)) return 1;
    else return fibonacci(n-1)+fibonacci(n-2);
  }

  public static void main (String [] args) {
    int maxzahl = Integer.parseInt(args[0]);
    for (int i = 1; i<=maxzahl; i++) {
      System.out.println(i+": " + fibonacci(i));
    }
  }
}
```

4. Ulam-Zahlen

Stanislaw Marcin Ulam, ein polnischer Mathematiker, beschäftigte sich mit folgendem Problem (auch „Collatz-Problem“). Zunächst soll eine Zahlenfolge folgendermaßen definiert werden:

$a_{n+1} = a_n / 2$ (falls a_n gerade ist) oder

$a_{n+1} = 3 \cdot a_n + 1$ (falls a_n ungerade ist).

Interessanterweise entwickelt sich die Folge a_n nach 1. Die noch immer ungelöste Vermutung lautet, dass dies für jede natürliche Zahl gilt.

```
public class Ulam{
    static int ulam(int n) {
        System.out.println(n);
        if (n==1) return 1;
        else {
            if ((n % 2) == 0) return ulam(n/2);
            else return ulam(3*n+1);
        }
    }

    public static void main(String [] args) {
        int maxzahl = Integer.parseInt(args[0]);
        ulam(maxzahl);
    }
}
```

Führen wir dieses Programm für verschiedene Startwerte a_1 aus, so erhalten wir beispielsweise:

```
nus@ice:~/java_bsp> java Ulam 5
5
16
8
4
2
1
```

Offenbar führt die Folge sicher zum Endwert 1, wenn irgendein Folgenglied eine Potenz zur Basis 2 erreicht.

```
nus@ice:~/java_bsp> java Ulam 6
6
3
10
5
16
8
4
2
1
```

5. Die Hofstadter-Funktion

Nach der Definition

h(1) = 1
h(2) = 1
h(n) = h(n – h(n-1)) + h(n – h(n-2))

lassen sich sehr einfach Funktionswerte berechnen, die stets gleich oder nur geringfügig größer als die Hälfte des Arguments n sind:

```
public class Hofstadter {

  public static int hof(int n) {
    if (n > 2) {
      return (hof(n - hof(n - 1)) + hof(n - hof
(n - 2)));
    } else {
      return 1;
    }
  }

  public static void main (String args[]) {
    int maxzahl = Integer.parseInt(args[0]);
    System.out.println(hof(maxzahl));
  }
}
```

Bei höheren Zahlen zeigt sich sofort, dass eine iterative Implementierung der Hofstadter-Funktion wesentlich effizienter läuft – während die rekursiv definierte Hofstadter-Funktion schon bei Argumenten ab etwa n = 50 Ergebnisse nicht mehr in vertretbaren Zeiten liefert, bereiten mit der nicht rekursiven Berechnung Argumente größer 10000 keine Schwierigkeiten (für 20000 liefert die Hofstadter-Funktion den Wert 10199):

```
public class Hofstadter {
  public static void main (String [] args) {
        int maxzahl = Integer.parseInt(args[0]);
        int [] hof = new int [maxzahl + 1];
        hof[1] = 1;
        hof[2] = 1;

        System.out.println("1: " + hof[1]);
        System.out.println("2: " + hof[2]);

        for (int i = 3; i<=maxzahl; i++) {
          hof[i] = hof[i - hof[i-1]] + hof[i -
hof[i-2]];
          System.out.println(i + ": " + hof[i]);
        }
  }
}
```

Sie werden sich fragen, was den (gewaltigen) Laufzeitunterschied bedingt. Beim Aufruf einer Methode müssen bestimmte Informationen zwischenzeitlich gespeichert werden (Rücksprungadressen, Werte lokaler Variablen, Funktionsergebnisse). Dies geschieht in einer Weise, die sicher stellt, dass der zuletzt gespeicherte Wert wieder als erster ausgelesen wird. Dieses Prinzip, „Last in – First out“, wird mit einem Stapelspeicher („Stack“) realisiert. Die Zugriffe auf diesen Stack verlangsamen die rekursive Lösung mit zunehmender Rekursionstiefe immer mehr, während bei der nicht rekursiven Implementierung lediglich (sehr rasche) Zugriffe auf die Speicherplätze eines Arrays notwendig sind.

Im Allgemeinen laufen iterative Methoden bedeutend rascher als rekursive Methoden.

6. Die Ackermann-Funktion

Der Mathematiker Wilhelm Ackermann definierte eine Folge, deren Glieder selbst Funktionen aus den vorhergehenden Gliedern bilden (vgl. [7]). Die von der ungarischen Mathematikerin Rósza Péter vereinfachte Version verwenden wir für das nächste Beispiel. Dabei ist definiert:

a(0,m) = m + 1
a(n,0) = a(n-1,1)
a(n,m) = a(n-1, a(n, m-1))

```
public class Ackermann {
  static long ackermann(long n, long m) {
        if (n==0) return m+1;
        else if (m==0) return ackermann(n-1,1);
        else return ackermann(n-1, ackermann
(n,m-1));
  }

  public static void main (String [] args) {

    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);

    System.out.println("a(" + x + "," + y + ") =
" + ackermann(x,y));
  }
```

```
}
```

Vorsicht: Beim Berechnen der Ackermann-Funktion stellt sich heraus, dass die Berechnungen sehr aufwändig werden. Bereits bei kleinen Argumenten führen die Rekursionen zu einem so genannten „Stacküberlauf“ - die Rekursionstiefe wurde zu groß.

```
nus@ice:~/java_bsp> java Ackermann 3 10
a(3,10) = 8189
nus@ice:~/java_bsp> java Ackermann 4 1
Exception in thread "main"
java.lang.StackOverflowError
```

7. Legendär: Die Türme von Hanoi

Die (legendären) Türme von Hanoi bestehen aus durchbohrten Scheiben, die der Größe nach auf Säulen geschlichtet sind. Auf einer Säule befinden sich n Scheiben - sie sollen Stück für Stück auf eine andere Säule umgeschlichtet werden, wobei eine Scheibe immer nur auf einer größeren Scheibe zu liegen kommen darf. Damit dies möglich ist, können Scheiben zwischenzeitlich auf einer dritten Säule gestapelt werden - aber auch auf dieser Säule darf stets eine Scheibe nur auf einer größeren zu liegen kommen.

Im folgenden Beispiel wird das Umschlichten der Scheiben rekursiv berechnet und Schritt für Schritt ausgegeben:

```
public class Hanoi {

  static long schritte;

  static void lege(int n, char von, char nach,
char zwischen) {
    if (n > 0) {
      lege (n-1, von, zwischen, nach);
      System.out.println(n + ". Scheibe von " +
von + " nach " + nach);
      lege (n-1, zwischen, nach, von);
      schritte ++;
    }
  }

  public static void main (String [] args) {
    int maxzahl = Integer.parseInt(args[0]);
    lege (maxzahl, 'a', 'c', 'b');
    System.out.println
("---------------------------------------");
    System.out.println(schritte + " Schritte");
  }
}
```

Damit erhalten wir für 4 Scheiben beispielsweise folgende „Spielzüge“:

```
nus@ice:~/java_bsp> java Hanoi 4
1. Scheibe von a nach b
2. Scheibe von a nach c
1. Scheibe von b nach c
3. Scheibe von a nach b
1. Scheibe von c nach a
2. Scheibe von c nach b
1. Scheibe von a nach b
4. Scheibe von a nach c
1. Scheibe von b nach c
2. Scheibe von b nach a
1. Scheibe von c nach a
3. Scheibe von b nach c
1. Scheibe von a nach b
2. Scheibe von a nach c
1. Scheibe von b nach c
---------------------------------------
15 Schritte
```

Mit 4 Münzen lässt sich dies sehr rasch „nachspielen“. Wie viele Schritte sind für 5, 6, ... Scheiben erforderlich? Wie lange dauert das Umschlichten, wenn man für das Umlegen einer Scheibe 2 Sekunden benötigt?

8. Ausblick, Übungen

Rekursive Algorithmen werden in allen Programmierumgebungen verwendet.

- Formulieren Sie eine rekursive Funktion zur Berechnung von Potenzen $f(x) = x^k$!
- Berechnen Sie die Summe $1 + x^{-1} + x^{-2} + ... + x^{-n}$ rekursiv und iterativ!
- Formulieren Sie ein Programm zur Berechnung der Fibonaccizahlen, das ohne rekursive Methodenaufrufe auskommt!
- Untersuchen Sie das Verhalten der Funktionswerte, wenn Sie die Fibonaccifolge in einer Restklasse (zB. mod 7) berechnen!
- Der größte gemeinsame Teiler kann mit der folgenden rekursiv aufgerufenen Methode berechnet werden:

```
static int ggT(int a, int b) {
  if (a == b) return a;
  else if (a < b) return ggT(a,b-a);
  else return ggT(a-b,b);
}
```

  Formulieren Sie eine entsprechende JAVA-Klasse!

- Mit Rekursionen lassen sich ansprechende Grafiken erzeugen. Recherchieren Sie dazu Beispiele aus der fraktalen Geometrie (zB [1])!

8. Literatur, Weblinks

[1] PC-News Nr. 76, Feb. 2002, S. 49, „JAVA - Fraktale“

[2] PC-News Nr. 98, , S. 25, „JAVA – Verschiedene Ausgaben auf der Konsole“

[3] Guido Krüger, „Handbuch der JAVA-Programmierung“, Addison & Wesley, ISBN 3-8273-2201-4

[4] http://www.javabuch.de („Handbuch der JAVA-Programmierung“, freier Download für schulische Zwecke)

[5] Herbert Schildt, „JAVA – Grundlagen der Programmierung“, mitp, IBSN 3-8266-1524-7

[6] Udo Müller, „JAVA – das Lehrbuch“, mitp, ISBN 3-8266-1333-3

[7] Christian Ullenboom, „Java ist auch eine Insel“, Galileo Computing, ISBN 3-89842-365-4

[8] http://www.gymmelk.ac.at/nus/informatik/wpf/JAVA (Unterrichtsbeispiele zum Programmieren mit JAVA, Quelltexte zum Downloaden)