Rekursion - Teil III

Bei der rekursiven Lösung zur Berechnung der *n*-ten Fibonacci-Zahl (s. u.) werden keine Variablen benötigt und das Programm ist deutlich kürzer und übersichtlicher. Das liegt an der rekursiven Struktur der mathematischen Definition, die eine rekursive Problemlösung geradezu fordert.

```
public class Fibonacci {

  public static long fibIterativ(int n){
    long fib_n_minus_2 = 1;
    long fib_n_minus_1 = 1;
    long fib_n = 0;

    switch(n){
      case 1: return 1;
      case 2: return 1;
      default:
        for (int i=0;i<n-2;i++) {                          // Iteration
          fib_n         = fib_n_minus_1 + fib_n_minus_2;
          fib_n_minus_2 = fib_n_minus_1;
          fib_n_minus_1 = fib_n;
        }
        return fib_n;
    }
  }
```

```
  public static long fibRekursiv(int n){
    switch(n){
      case 1:  return 1;                                   // Abbruch
      case 2:  return 1;                                   // Abbruch
      default: return fibRekursiv(n-1) + fibRekursiv(n-2); // Rekursion
    }
  }
```

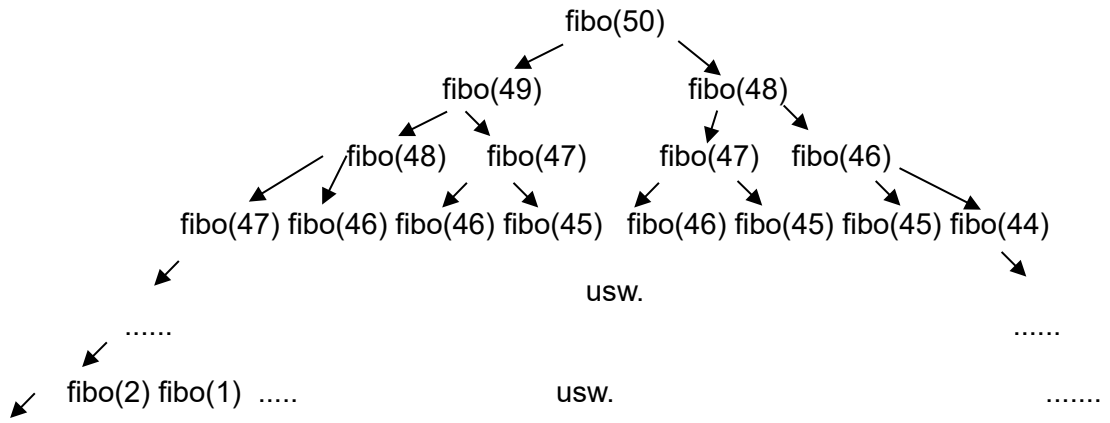
```
  public static void main(String[] args) {

     System.out.println("Dieses Programm berechnet die n-te Fibonacci-Zahl.\n");
     int n = IOTools.readInt("Geben Sie die Zahl n ein: ");
     System.out.println();

     if(n>0){
       System.out.println("Die "+n+"-te Fibonacci-Zahl lautet (iterativ): "
                                                              +fibIterativ(n));
       System.out.println("Die "+n+"-te Fibonacci-Zahl lautet (rekursiv): "
                                                              +fibRekursiv(n));
     }else{
       System.out.println("\nFalsche Eingabe!");
     }
  }
}
```

Bei genauerer Betrachtung zeigt sich jedoch, dass der rekursive Ansatz einen erheblichen Speicherbedarf verursacht und das Programm für große *n* praktisch unbrauchbar ist. Starte einmal die iterative und die rekursive Variante mit *n*=40 und *n*=50 und betrachte das Ergebnis.

Die iterative Lösung liefert das Ergebnis sofort und bei der rekursiven Lösung dauert es etwa 5 Minuten!!! Woran das liegt, werden wir nun genauer beleuchten.

Die iterative Lösung muss pro Schleifendurchlauf 3 Zuweisungen und eine Additionen durchführen, wenn man den Aufwand für die for-Schleife einmal vernachlässigt. Bei n=50 sind das etwa 4*50 =200 Operationen. Natürlich ist das Programm sofort fertig. Allgemein lässt sich sagen, dass der iterative Algorithmus den Aufwand O(n) hat, also einen linearen Aufwand.

Bei der rekursiven Lösung erzeugt der Aufruf `fib(50)` zwei neue Funktionsaufrufe `fib(49)` und `fib(48)`, wobei jeder von den beiden wieder zwei neue Aufrufe erzeugt usw.



Versuchen wir einmal, die Anzahl der Aufrufe abzuschätzen. Der Baum hat links die Höhe *n* und rechts die Höhe *n/2*. Da ein voller Baum der Höhe *h* stets 2^h-1 Knoten besitzt, ist die Anzahl der Funktionsaufrufe größer als $2^{n/2}$- 1 und kleiner als 2^n- 1. Also $2^{n/2}$- 1 < A (n) < 2^n- 1. In jedem Fall wächst der Aufwand aber proportional zu 2^n und damit exponentiell, also O(2^n).

Beispiel:

Geben wir z.B. *n = 100* in unser rekursives Programm ein, so werden wir vor eine harte Geduldsprobe gestellt. Es stellt sich die Frage, ob der Computer noch rechnet oder ob er abgestürzt ist. Nach obiger Abschätzung muss der Rechner mindestens 2^{50}- 1 Funktionsaufrufe durchführen, d.h. er muss jeweils die aktuelle Variable auf den Stack legen, neue Parameter berechnen und zuweisen und die Rücksprungadresse abspeichern, damit er weiß, wo er nach der Beendigung des Methodenaufrufs die zuletzt gültigen Werte vorfindet. Dazu kommt noch das Prüfen der Bedingung in der Fallunterscheidung und die Addition. Sagen wir also grob 6 Operationen pro Funktionsaufruf. Damit ergibt sich ein Aufwand von mindestens $6*(2^{50}- 1) = 6 * 1{,}1258999*10^{15} = 6{,}7553994*10^{15}$ Operationen.

Ein Core 2 Duo Prozessor macht etwa 27 MIPS (million instructions per second). Damit ergäbe sich eine minimale Laufzeit von 250199979 Sekunden = 69499 Stunden = 2895 Tagen = 8 Jahren.

Fazit:

a) Algorithmen mit exponentieller Laufzeit $A(n) = c^n$ sind nicht praktikabel.

b) Rekursion ist ein elegantes Konstruktionsmittel, es sollte jedoch nur mit Vorsicht angewandt werden. Schließlich hat z. B. hier die iterative Version nur einen linearen Aufwand A(n) = n.

c) Es gibt sogar eine Formel, um die *n*-te Fibonacci-Zahl direkt zu berechnen, also A (n) = c. Es lohnt sich also auch hier, im Vorfeld etwas Mathematik zu betreiben!