Dr. Jürg M. Stettbacher
Neugutstrasse 54
CH-8600 Dübendorf

Telefon: +41 43 299 57 23
Email: dsp@stettbacher.ch

Übung

Algorithmen

Betrachten Sie diese Zahlenfolge:

1 1 2 3 5 8 13 21 ...

Die einzelnen Zahlen F seien nummeriert, beginnend bei null[1]. Demnach ist:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_2 &= 2 \\ F_3 &= 3 \\ F_4 &= 5 \\ F_5 &= 8 \\ &... \end{aligned}$$

1. Bestimmen Sie die drei Zahlen F_m $(m = 8 \dots 10)$ der Folge.
2. Wie heisst die Zahlenfolge?
3. Beschreiben Sie in Worten das rekursive Bildungsgesetz für jede Zahl F_n mit $n \geq 0$.
   Vergessen Sie die Anfangswerte nicht.
4. Notieren Sie in mathematischer Schreibweise das rekursive Bildungsgesetz für jede Zahl F_n mit $n \geq 0$.
5. Notieren Sie, ausgehend von der mathematischen Beschreibung, einen rekursiven Algorithmus, der für einen gegebenen Index n die entsprechende Zahl F_n erzeugt.
6. Entwickeln Sie ein Computer-Programm, das den oben beschriebenen, rekursiven Algorithmus implementiert. Der Index n soll dabei als Parameter dem Programm übergeben werden. Anschliessend soll das Programm alle Zahlen F_k mit $k = 0 \dots n$ auflisten.
7. Zeigen Sie mit dem Programm alle Zahlen F_k mit $k = 0 \dots 100$ an. Welches Problem tritt auf?
   Versuchen Sie, den Verlauf des Programms in geeigneter Weise grafisch darzustellen und machen Sie eine Abschätzung für die Berechnungsdauer.
8. Implementieren Sie das Programm in einer alternativen Weise, so dass die Probleme nicht mehr auftreten.
9. Was passiert nun, wenn Sie die Zahlen F_k von $k = 0 \dots 500$ berechnen lassen?

[1] Ohne es speziell zu erwähnen setzen wir voraus, dass die Zahlen F natürliche Zahlen sind, also Elemente der Menge $\mathbb{N}$. Ebenso sind die Indices natürliche Zahlen aus der Menge $\mathbb{N}_0$. Im Gegensatz zu $\mathbb{N}$ schliesst die Menge $\mathbb{N}_0$ die Zahl Null mit ein.

Antworten

1. Wir können die folgende Fortsetzung erraten:

$$\begin{aligned} F_8 &= 34 \\ F_9 &= 55 \\ F_{10} &= 89 \end{aligned}$$

2. Die Zahlenfolge heisst *Fibonacci-Folge*[2].

3. Die ersten beiden Werte F_0 und F_1 der Folge sind eins. Alle weiteren sind jeweils die Summe der beiden Vorgänger in der Folge.

4. Mathematisch ausgedrückt:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{für } n \geq 2 \end{aligned}$$

   Oder kompakter:

$$F_n = \begin{cases} 1 & \text{für } n = 0, 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$

5. Algorithmus FIB(n):

   (a) Falls $n < 0$: Beende Algorithmus mit einer Fehlermeldung.

   (b) Falls $n = 0$: Beende Algorithmus und gib den Wert 1 zurück.

   (c) Falls $n = 1$: Beende Algorithmus und gib den Wert 1 zurück.

   (d) Falls der Algorithmus hier anlangt, ist $n \geq 2$:
   Beende Algorithmus und gib den Wert FIB(n-1)+FIB(n-2) zurück.

   Beachte: Die Punkte (b) und (c) könnten auch zusammen gefasst werden. Der Punkt (d) ist aufwändiger als es scheint, denn der Rückgabewert FIB(n-1)+FIB(n-2) muss zuerst durch erneute Aufrufe des selben Algorithmus berechnet werden.

6. In Java könnte der Algorithmus so aussehen:

```
// File 'Fibonacci.java'.
public class Fibonacci {

        // Function FIB(k):
        public static long fib(int k) {
                // Fibonacci for sequence numbers 0 and 1:
                if (k == 0) return 1;
                if (k == 1) return 1;
                // Fibonacci for sequence numbers >= 2:
                return fib(k-1)+fib(k-2);
        }

        // Main function and starting point:
        public static void main(String[] args) {

                // Read input parameter n from command line:
                int n = Integer.parseInt(args[0]);
```

[2] Die Fibonacci-Folge hat keine spezielle Bedeutung, ausser dass sie gewisse Wachstumsprozesse der Natur nachbildet und jeweils zwei aufeinander folgende Zahlen den Goldenen Schnitt approximieren. Der Goldene Schnitt ist ein Teilungsverhältnis von Strecken, das als besonders ästhetisch gilt.

```
                // Check for invalid input parameter:
                if (n < 0) {
                        System.out.println("Fibonacci number does not exist!");
                        System.exit(0);
                }

                // Produce all Fibonacci numbers k up to n:
                for (int k=0; k<=n; k++)
                        System.out.println(k + ": " + fib(k));
        }
}
```

Auf einer Kommandozeile lässt sich das Programm so übersetzen[3]:

```
>> javac Fibonacci.java
```

Nach erfolgreichem Übersetzen kann man mit dem folgenden Befehl auf einer Kommandozeile die ersten zehn Elemente der Fibonacci-Folge berechnen:

```
>> java Fibonacci 10
```

In der Programmiersprache C könnte der Algorithmus so ausschauen:

```
// File 'fibonacci.c'.

#include <stdio.h>
#include <stdlib.h>

// Function FIB(k):
long fib(int k) {

        // Test k:
        if (k < 2) {
                return(1);
        } else {
                return(fib(k-1)+fib(k-2));
        }
}

// Main function and entry point:
int main(int argc, char *argv[]) {

        long k, n;

        // Get command-line argument:
        n = atol(argv[1]);

        // Check command line argument:
        if (n < 0) {
                printf("ERROR: invald n!\n");
                return(-1);
        }

        // Number n is valid, print all Fibonacci numbers up to n:
        for (k=0; k<=n; k++) {
                printf("%3ld: %4ld\n", k, fib(k));
        }

        return(0);
}
```

Auf einer Kommandozeile lässt sich das Programm so übersetzen[4]:

```
>> gcc fibonacci.c -o fibonacci
```

Nach erfolgreichem Übersetzen kann man mit dem folgenden Befehl auf einer Kommandozeile die ersten zehn Elemente der Fibonacci-Folge erzeugen:

[3] Es muss auf dem Rechner das JDK (Java Development Kit) und das JRE (Java Runtime Environment) installiert sein. Unter Windows muss allenfalls die PATH-Variable noch ergänzt werden. Typischerweise heisst der Pfad wie folgt: Für x86 "C:\program files (x86)\java\jdkXXX\bin", für x64 "C:\program files\java\jdkXXX\bin". Dabei steht XXX für die Java-Version.

[4] Es wird hier voraus gesetzt, dass ein GNU C-Compiler *gcc* auf dem Rechner installiert ist, wie das typischerweise bei Linux der Fall ist. Das Äquivalent für Windows heisst MinGW (Minimal GNU for Windows).

```
>> ./fibonacci 10
```

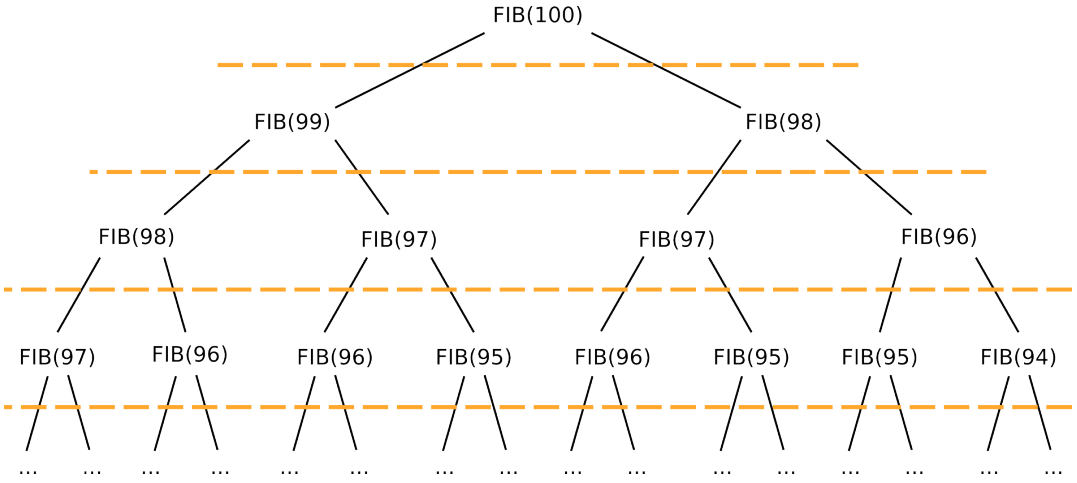
7. Das Java-Programm wird folgendermassen aufgerufen:

   ```
   >> java Fibonacci 100
   ```

   Der Aufruf des C-Programms sieht so aus:

   ```
   >> ./fibonacci 100
   ```

   Es zeigt sich, dass das Berechnen der Zahlen F_k schon bald sehr lange dauert. Nach einigem Warten vermuten wir, dass die Zahl F_{100} wohl ewig nicht erscheinen wird. Die folgende Skizze zeigt, was abläuft:



   Der Aufruf von FIB(100) zerfällt in zwei weitere Aufrufe, FIB(99) und FIB(98). Jeder von diesen erzeugt zwei weitere Aufrufe der Funktion FIB(.), usw. Auf der obersten Ebene haben wir demnach mit FIB(100) nur $2^0 = 1$ Aufruf. Auf der zweiten Ebene sind es $2^1 = 2$ Aufrufe, auf der dritten Ebene $2^2 = 4$ Aufrufe, etc. Total gibt es also rund 100 Ebenen, wobei die untersten nicht mehr voll belegt sind. Dies wollen wir aber für die Aufwandschätzung ignorieren. Wir gehen also von 100 Ebenen aus, wobei die letzte Ebene $2^{99} = 6.3 \cdot 10^{29}$ Aufrufe enthält. Für die gesamte Berechnung von FIB(100) sind demnach rund $2^{100} = 1.3 \cdot 10^{30}$ Aufrufe der Funktion FIB(.) nötig.

   Angenommen, die Behandlung eines Aufrufs von FIB(.) dauere 1 ns (das Entspricht einer Rate von 1 GHz), so dauert die gesamte Berechnung $1.3 \cdot 10^{21}$ Sekunden. Dies entspricht rund $4.0 \cdot 10^{13}$ Jahre, was ein Vielfaches des vermuteten Alters des Universums von ca. 13 Mia. Jahre ist ($13 \cdot 10^9$ Jahre). Wir brauchen also nicht auf das Resultat von FIB(100) zu warten.

   Unser rekursiver Algorithmus ist sehr elegant aufgebaut. Er verursacht aber einen unermesslichen Aufwand. In der Abbildung erkennen wir sofort, dass viele FIB(p) mit $p < 100$ mehrfach rekursiv berechnet werden, was für diesen Aufwand verantwortlich ist. Selbstverständlich tritt dieses Problem nicht bei jedem rekursiven Algorithmus auf.

8. Das oben erkannte Problem lässt sich leicht beheben, indem FIB(100) nicht durch eine Kaskade von rekursiven Aufrufen berechnet wird, sondern linear, ausgehend von F_0 und F_1. Der lineare Algorithmus FIBL(n) könnte so aussehen:

   (a) Falls $n < 0$: Beende Algorithmus mit einer Fehlermeldung.
   (b) Falls $n = 0$ oder $n = 1$: Beende Algorithmus mit dem Resultat 1.
   (c) Setze $A = 1$, $B = 1$, $C = A + B$ und $k = 2$.
   (d) Solange $k \neq n$, berechne:

$$\begin{aligned} A &= B \\ B &= C \\ C &= A + B \\ k &= k + 1 \end{aligned}$$

(e) Beende Algorithmus mit dem Resultat C.

Dieser Algorithmus weist keine Rekursionen mehr auf. Statt dessen beinhaltet er eine Schleife, in welcher k jeweils inkrementiert wird.

In Java könnte die Implementation so aussehen:

```
// File 'Fibonacci2.java'.
public class Fibonacci2 {

        // Function FIBL(n):
        public static long fibl(int n) {

                // Fibonacci for sequence numbers 0 and 1:
                if (k == 0) return 1;
                if (k == 1) return 1;
                // Fibonacci for sequence numbers >= 2:
                long A = 1; long B = 1; long C = A+B; int k = 2;
                while (k != n) {
                        A = B; B = C; C = A+B; k = k+1;
                }
                return C;
        }

        // Main function and starting point:
        public static void main(String[] args) {

                // Read input parameter n from command line:
                int n = Integer.parseInt(args[0]);

                // Check for invalid input parameter:
                if (n < 0) System.out.println("Fibonacci number does not exist!");

                // Produce all Fibonacci numbers k up to n:
                for (int k=0; k<=n; k++)
                        System.out.println(k + ": " + fibl(k));
        }
}
```

Das Pendant in C wäre etwa dies:

```
// File 'fibonacci2.c'.

#include <stdio.h>
#include <stdlib.h>

// Function FIBL(n):
long fibl(int n) {

        long A, B, C;
        int k;

        // Test n:
        if (n < 2) {
                return(1);
        } else {
                A = 1; B = 1; C = A+B; k = 2;
                while (k != n) {
                        A = B; B = C; C = A+B; k = k+1;
                }
                return(C);
        }
}

// Main function and entry point:
int main(int argc, char *argv[]) {

        long k, n;

        // Get command-line argument:
        n = atol(argv[1]);

        // Check command line argument:
        if (n < 0) {
                printf("ERROR: invald n!\n");
                return(-1);
        }
```

```
        // n is valid, print all Fibonacci numbers up to n:
        for (k=0; k<=n; k++) {
                printf("%3ld: %4ld\n", k, fibl(k));
        }

        return(0);
}
```

9. Werden mit diesen Programmen viele Fibonacci-Zahlen berechnet, so stellt man plötzlich fest, dass negative Zahlen auftreten, zum Beispiel so wie hier:

```
83: 160500643816367088
84: 259695496911122585
85: 420196140727489673
86: 679891637638612258
87: 1100087778366101931
88: 1779979416004714189
89: 2880067194370816120
90: 4660046610375530309
91: 7540113804746346429
92: -6246583658587674878
93: 1293530146158671551
94: -4953053512429003327
95: -3659523366270331776
```

Negative Werte dürften aber nie auftreten, da jede Fibonacci-Zahl aus der Summe von zwei positiven Vorgängern besteht. Um den Fehler zu verstehen muss man bedenken, dass jeder Prozessor[5] und jede Programmiersprache[6] für Zahlen einen bestimmten Speicherbereich vorsieht. Im oben dargestellten Fall handelt es sich um Zahlen mit 64 Bit Breite (8 Byte)[7]. Die grösste darstellbare Zahl mit Vorzeichen in 64 Bit ist $2^{63} - 1 \approx 9.2 \cdot 10^{18}$. Dies ist eine Zahl mit total 19 Stellen. In der oben abgebildeten Liste erkennt man, dass die 92. Fibonacci-Zahl diesen Bereich überschreitet. Das heisst, es entsteht auf der linken Seite ein Überlauf im Register, welcher verloren geht. Der Rest, der im Register stehen bleibt, entspricht dann (zufällig) einer negativen Zahl. Der Prozessor registriert den Überlauf zwar in seinem Carry-Flag, das Programm ignoriert jedoch das Flag[8]. So bleibt der Rechenfehler unerkannt und alle folgenden Fibonacci-Zahlen sind falsch.

[5] Bei Mikroprozessoren entspricht das Zahlenformat normalerweise der Registerbreite.

[6] Programmiersprachen können Zahlenformate emulieren, die von der Registerbreite des verwendeten Mikroprozessors abweichen. In diesem Fall verursacht aber jede Berechnung einen höheren Rechenaufwand.

[7] Auf einer x86_64 Architektur ist der C-Datentyp *long* 64 Bit breit. In Java hat er generell 64 Bit. Ein *int* umfasst dagegen in beiden Fällen 32 Bit.

[8] Viele Programmiersprachen kennen keine Mechanismen um auf ein gesetztes Carry-Flag zu reagieren. Es bleibt daher dem Programmierer überlassen, allfällige Tests vor der Berechnung durchzuführen und allenfalls präventiv zu reagieren. Verzichtet er darauf, so muss er in Kauf nehmen, dass falsche Resultate auftreten können.