Kapitel 8

Rekursion

Ein Objekt heißt *rekursiv*, wenn es sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist. Rekursion kommt nicht nur in der Mathematik, sondern auch im täglichen Leben vor. Wer hat etwa noch nie Bilder gesehen, die sich selbst enthalten?

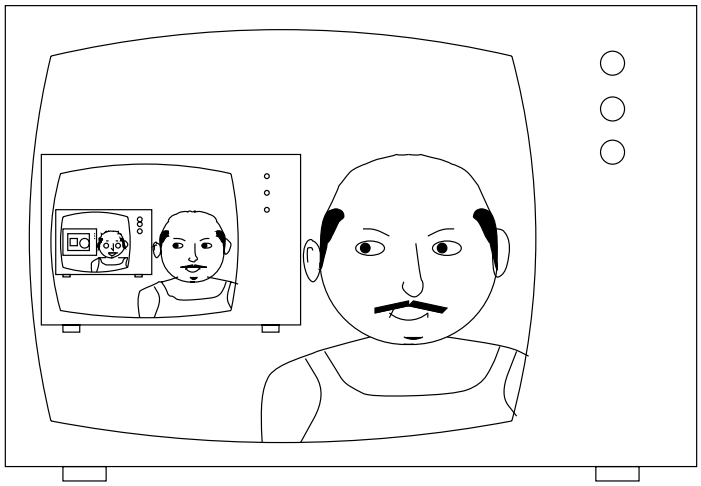

Abbildung 8.1: Rekursion im Bild.

Rekursion kommt speziell in mathematischen Definitionen zur Geltung. Bekannte Beispiele sind die natürlichen Zahlen, Baumstrukturen und gewisse Funktionen:

1. Natürliche Zahlen

   Die Menge $\mathbb{N}$ der natürlichen Zahlen kann wie folgt rekursiv definiert werden.

   - $0 \in \mathbb{N}$
   - Ist $n \in \mathbb{N}$, so auch der *Nachfolger* $n+1$ von n.

2. Binäre Bäume

Binäre Bäume sind gerichtete Bäume (vgl. Seite 122), bei denen jeder Knoten Anfangspunkt von höchstens zwei Kanten ist. Die Menge der binären Bäume kann wie folgt rekursiv definiert werden (Übung).

- Der leere Baum ohne Knoten ist ein binärer Baum (genannt der *leere Baum*).
- Sind T_1 und T_2 binäre Bäume, so ist auch der Graph bestehend aus einer Wurzel w und den Teilbäumen T_1 und T_2 ein binärer Baum, vgl. Abbildung 8.2. Ist einer der Teilbäume leer, so fehlt die zu ihm führende Kante.

  Die nichtleeren binären Bäume mit höchstens 3 Knoten sind in Abbildung 8.3 dargestellt.

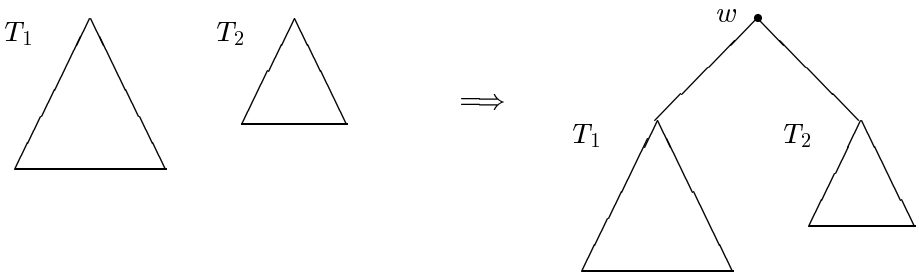

Abbildung 8.2: Rekursion bei binären Bäumen.

Abbildung 8.3: Die nichtleeren binären Bäume mit höchstens 3 Knoten.

3. Die Fakultät $n!$ einer natürlichen Zahl n kann rekursiv definiert werden als

$$n! = \begin{cases} 1\,, & \text{falls } n = 0 \\ n \cdot (n-1)\,, & \text{falls } n > 0 \end{cases}$$

Das Wesentliche der Rekursion ist die Möglichkeit, eine *unendliche* Menge von Objekten durch eine *endliche* Aussage zu definieren. Auf die gleiche Art kann eine unendliche Zahl von Berechnungen durch ein endliches rekursives Programm beschrieben werden, ohne daß das Programm explizite Schleifen enthält. Rekursive Algorithmen sind hauptsächlich dort angebracht, wo das Problem, die Funktion oder die Datenstruktur bereits rekursiv definiert ist.

Ein notwendiges und hinreichendes Werkzeug zur Darstellung rekursiver Programme sind Unterprogramme (also in C++ Funktionen), die sich selbst oder gegenseitig aufrufen können. Enthält eine Funktion `f()` einen expliziten Aufruf ihrer selbst, so heißt `f()` *direkt rekursiv*; enthält `f()` einen Aufruf einer zweiten Funktion `g()`, die dann ihrerseits `f()` (direkt oder indirekt) aufruft, so heißt `f()` *indirekt rekursiv*. Das Vorhandensein von Rekursion muß daher nicht direkt aus der Funktion ersichtlich sein.

Wie Wiederholungsanweisungen bergen auch rekursive Funktionen die Gefahr nicht abbrechender Ausführung und verlangen daher die Betrachtung des Problems der *Terminierung*. Grundlegend ist daher die Bedingung, daß der rekursive Aufruf einer Funktion von einer Bedingung B abhängt, die irgendwann nicht mehr erfüllt ist.

8.1 Beispiele für Rekursive Algorithmen

Zur Beachtung vorab: manchmal sollte man aus Komplexitätsgründen statt Rekursion lieber Iteration verwenden. Hierauf wird in Kapitel 8.2 ausführlich eingegangen.

8.1.1 Berechnung des ggT

Die Berechnung des ggT erfolgte in Beispiel 3.3 iterativ durch eine `while`-Schleife. Aufgrund des dort gezeigten Lemma 3.1 gilt auch die folgende rekursive Darstellung des ggT für natürliche Zahlen $x \geq y \geq 1$.

$$ggT(x,y) = \begin{cases} x\,, & \text{falls } y = 0 \\ ggT(y, x \bmod y)\,, & \text{falls } y > 0 \end{cases}$$

Diese rekursive Darstellung führt direkt zu dem folgenden rekursiven C++-Programm.

Programm 8.1 `ggTrekursiv.cc`

```
#include <iostream.h>

int ggT( /* in */ int a,
         /* in */ int b);
    //.....................................................
    // PRE:  a > 0 && b > 0
    // POST: FCTVAL == greatest common divisor of a and b
    //                 (algorithm: the Euclidean algorithm)
    //.....................................................

void main()
{
    int x, y;
    cout << "Bitte natuerliche Zahlen x und y zur Bestimmung von ggT(x,y)"
         << " eingeben.\nx,y > 0: ";
    cin >> x >> y;
    cout << "ggT(" << x << ',' << y << ") == " << ggT(x,y) << endl;
}

int ggT( /* in */ int a,
         /* in */ int b)
{
    int temp;
    if ( a < b ){
```

```
        temp = a; a = b; b = temp;
    }//endif
    temp = a % b;
    if ( temp == 0 ) return b;
    else return ggT(b,temp);
}
```

Der Aufrufbaum (Rekursionsbaum) ist in Abbildung 8.4 angegeben. Er entartet hier zur Liste, da keine Verzweigung bei den Aufrufen auftritt.

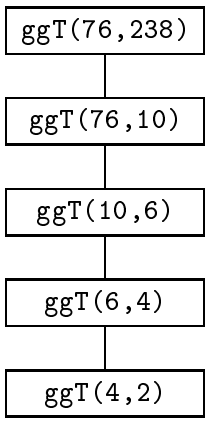

Abbildung 8.4: Der Rekursionsbaum von Programm 8.1.

Die Rekursionstiefe ist die Höhe des Rekursionsbaums plus 1, also 5. Der Run-Time Stack enthält daher beim "tiefsten" Aufruf 5 Activation Records, die in Abbildung 8.5 dargestellt sind.

Wir werden jetzt die Korrektheit des Programms beweisen und die maximale Rekursionstiefe abschätzen. Bei der Korrektheit unterscheiden wir zwischen *partieller Korrektheit* (beim Terminieren des rekursiven Algorithmus liegt das korrekte Resultat vor) und *totaler Korrektheit* (bei jedem Aufruf terminiert die Abarbeitung mit dem korrekten Resultat).

Satz 8.1 *Das Programm 8.1 berechnet den größten gemeinsamen Teiler korrekt. Ist* `ggT`(n, m) *ein Aufruf des Programms und ist* $M := \max\{n, m\}$*, so gilt für die Rekursionstiefe (und die Gesamtzahl der Aufrufe)* $R_{ggT}(n, m)$*:*

$$R_{ggT}(n, m) \leq 1 + 2 \cdot \log M$$

Beweis: Sei $x > y$ und sei $t := x$ div y und $r := x$ mod y. Dann gilt:

$$\begin{aligned} x &= t \cdot y + r \geq y + r \qquad (\text{da } t \geq 1) \\ &> r + r \qquad (\text{da } y > r) \end{aligned}$$

Also ist $r < \frac{x}{2}$. Somit wird in der Paarfolge (x, y), (y, r), $(r, \cdot)$ usw., die beim rekursiven Aufruf von `ggT` entsteht, das größere der beiden Elemente x, y nach zwei Aufrufen mehr als halbiert. Also kann es höchstens $1 + 2 \cdot \log M$ Aufrufe geben.

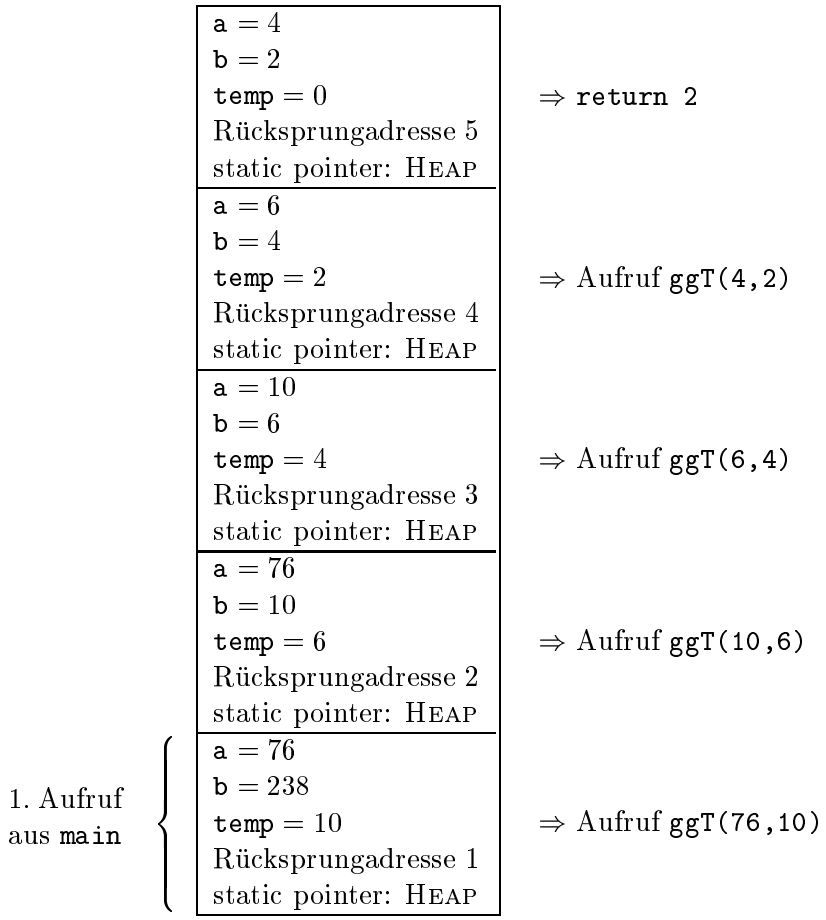

Abbildung 8.5: Der Run-Time-Stack von Programm 8.1.

Der Algorithmus terminiert also bei jedem Aufruf (sogar schnell). Die Korrektheit des dann gelieferten Ergebnisses folgt aus Lemma 3.1, da ja die Operation x mod y auf die fortgesetzte Subtraktion zurückgeführt werden kann. Die t-malige Anwendung von Lemma 3.1 auf $x = t \cdot y + r$ ergibt nämlich gerade $ggT(x, y) = ggT(y, r)$. □

8.1.2 Die Türme von Hanoi

Dieses Problem geht auf eine hinterindische Sage zurück: in einem im Dschungel verborgenen hinterindischen Tempel sind Mönche seit Beginn der Zeitrechnung damit beschäftigt, einen Stapel von 50 goldenen Scheiben mit nach oben hin abnehmendem Durchmesser, die durch einen goldenen Pfeiler in der Mitte zusammengehalten werden, durch sukzessive Bewegungen jeweils einer einzigen Scheibe auf einen anderen goldenen Pfeiler umzuschichten. Dabei dürfen sie einen dritten Pfeiler als Hilfspfeiler benutzen, müssen aber darauf achten, daß niemals eine Scheibe mit größerem Durchmesser auf eine mit kleinerem Durchmesser zu liegen kommt. Eine Lösung für 3 Scheiben ist in Abbildung 8.6 dargestellt.

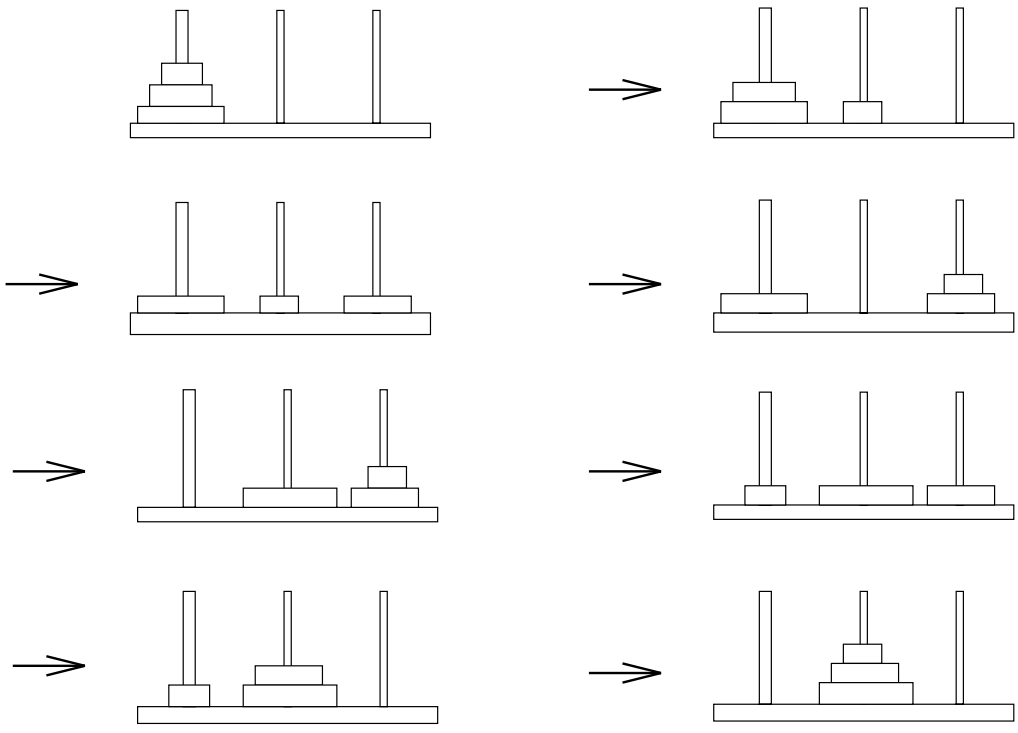

Abbildung 8.6: Umschichten von 3 Scheiben.

Die Sage berichtet, daß das Ende der Welt gekommen ist, wenn die Mönche ihre Aufgabe beendet haben.

Wir wollen uns nun ganz allgemein überlegen, wie man n Scheiben abnehmender Größe von einem Pfeiler i auf einen Pfeiler j ($1 \leq i,j \leq 3,\ i \neq j$) entsprechend der angegebenen Vorschrift umschichten kann.

Sei k der dritte zur Verfügung stehende Hilfspfeiler. Dann kann man das Problem, n Scheiben vom Pfeiler i zum Pfeiler j mit Hilfe des Pfeilers k umzuschichten, folgendermaßen lösen:

Man schichtet die obersten $n-1$ Scheiben vom Pfeiler i zum Pfeiler k mit Hilfe des Pfeilers j; dann bringt man die auf dem Pfeiler i verbliebene einzige (anfangs unterste) Scheibe (als einzige Scheibe) auf den Pfeiler j. Nun ist der Pfeiler i frei und man kann die $n-1$ Scheiben vom Pfeiler k auf den Pfeiler j mit Hilfe des Pfeilers i umschichten. Dies ist in Abbildung 8.7 dargestellt.

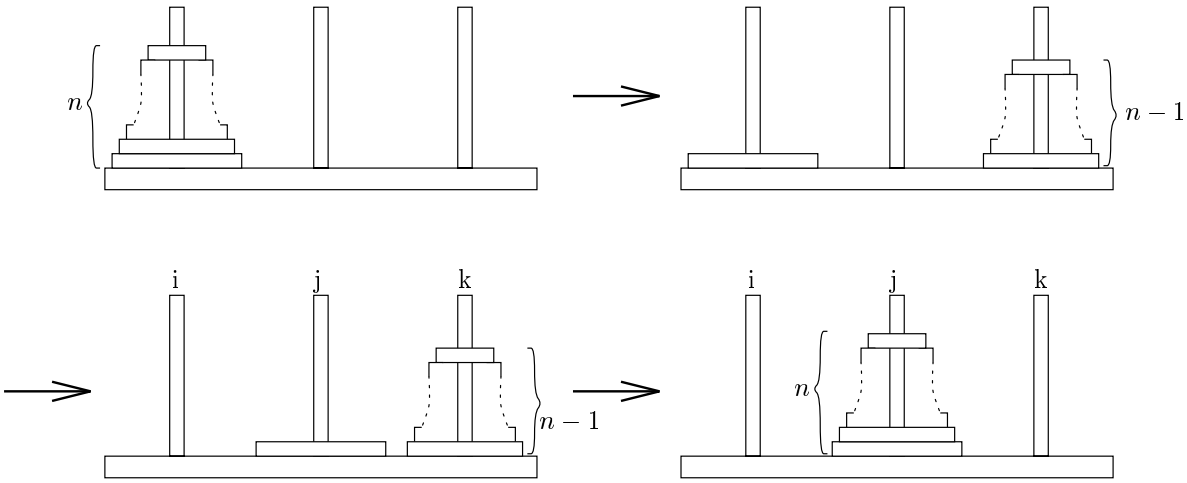

Abbildung 8.7: Umschichten von n Scheiben.

Wir definieren nun eine Funktion `Move()` so, daß der Aufruf `Move(n,i,j,k)` bewirkt, daß `n` Scheiben vom Pfeiler `i` zum Pfeiler `j` mit Hilfe des Pfeilers `k` so umgeschichtet werden, daß niemals eine größere auf eine kleinere Scheibe zu liegen kommt.

Der oben erläuterte rekursive Zusammenhang zwischen `Move(n,...)` und `Move(n-1,...)` führt dann zu folgender Funktion:

```
void Move( /* in */ int numberOfDisks,
           /* in */ int origin,
           /* in */ int destination,
           /* in */ int aux_pile )
    //.....................................................................
    // PRE:    numberOfDisks > 0 denotes the number of disks on origin
    //         && origin, destination, aux_pile are pairwise distinct with
    //            values in {1,2,3}
    //         && there are no disks on destination and aux_pile
    // POST:   numberOfDisks disks are on destination
    // EFFECT: writes the necessary moves on cout in the form
```

```
    //         i --> j, with i,j in {1,2,3}
    //.............................................................
{
    if ( numberOfDisks == 0 ) return; // nothing to do

    // move numberOfDisks - 1 smallest disks from origin to aux_pile
    // with destination as auxiliary file
    Move( numberOfDisks - 1, origin, aux_pile, destination);

    // write the move of the largest disk on cout
    cout << origin << " --> " << destination << endl;

    // move the numberOfDisks - 1 smallest disks from aux_pile
    // to destination with origin as auxiliary file
    // (the largest disk on destination does not interfere)
    Move( numberOfDisks - 1, aux_pile, destination, origin);
}
```

Diese Funktion führt bei Aufruf von `Move(3,1,2,3)` zu folgendem Output:

```
1 --> 2
1 --> 3
2 --> 3
1 --> 2
3 --> 1
3 --> 2
1 --> 2
```

Die rekursive Lösung ist hier besonders elegant, da der Rekursionsansatz einfach ist, während die Herleitung einer iterativen Lösung (die der Angabe einer expliziten "Strategie" zum Bewegen der Scheiben gleichkommt) wesentlich schwieriger ist (vgl. Übung).

Der Rekursionsbaum für den Aufruf von `Move(3,1,2,3)` ist in Abbildung 8.8 dargestellt. Er hat die Höhe 3. Also hat der Aufruf die Rekursionstiefe 4. Die Zahlen an den Aufrufen geben die Reihenfolge der Aufrufe an. Die zugehörige Belegung des Run-Time stack ist in Abbildung 8.9 wiedergegeben. Sie entspricht dem Durchlauf des Baumes in LRW-Ordnung.

Allgemein gilt:

Satz 8.2 *Beim Aufruf von* `Move(n,1,2,3)` *ist die Rekursionstiefe* $n+1$*, und die Gesamtanzahl der rekursiven Aufrufe* $2^{n+1}-1$*.*

Beweis: Übung. □

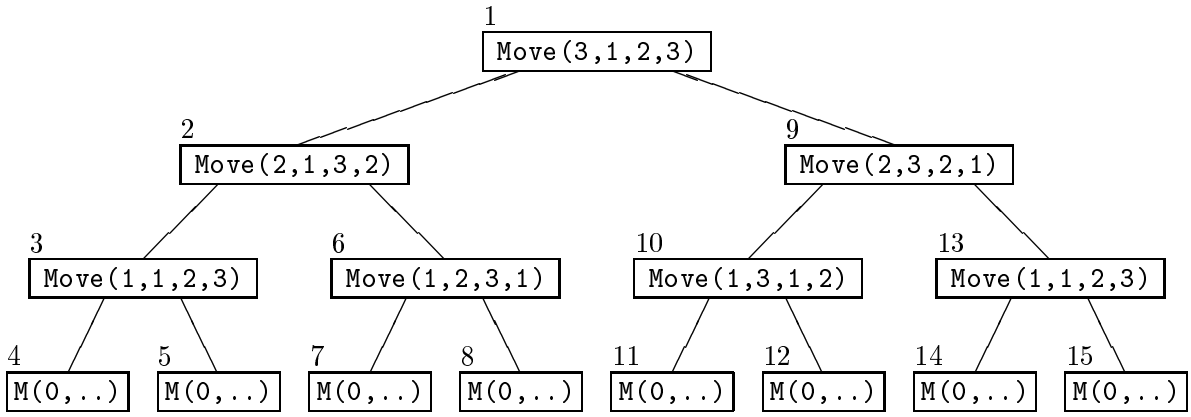

Abbildung 8.8: Rekursionsbaum zu den Türmen von Hanoi.

leer	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	usw.
		2	2	2	2	2	2	2	2	2	2	2	2		9	9	9	
			3	3	3	3	3		6	6	6	6	6			10	10	
				4		5				7		8					11	

Abbildung 8.9: Der Run-Time Stack zu `Move(3,1,2,3)`. Die Zahlen sind die Nummern der Aufrufe aus Abbildung 8.8

8.1.3 Die Ackermann Funktion

Als Beispiel für eine rekursive Funktionsdefinition komplexerer Art betrachten wir das Beispiel der Ackermann Funktion A, die als Extrapolation der Folge immer stärker wachsenden Funktionen Summe, Produkt, Potenz, ... aufgefaßt werden kann. Sie ist wie folgt definiert.

$$A(m,n) = \begin{cases} n+1\,, & \text{falls } m = 0 \\ A(m-1,1)\,, & \text{falls } m > 0,\ n = 0 \\ A(m-1, A(m,n-1))\,, & \text{falls } m,n > 0 \end{cases}$$

Auch diese Definition läßt sich unmittelbar in eine C++ Funktion übersetzen:

```
int A( /* in */ int m,
       /* in */ int n)
    //.........................................................
    // PRE:  m >= 0 && n >= 0
    // POST: FCTVAL == value of Ackermann function for m,n
    //.........................................................
{
    if ( m == 0 ) return n+1;
    else if ( n == 0 ) return A(m-1,1);
    else return A(m-1,A(m,n-1));
```

```
}
```

Für die Funktion A ist es bereits viel schwieriger zu sehen, wie (und daß überhaupt) jede Berechnung nach endlich vielen Schritten terminiert. Dies ist zwar der Fall, wie Satz 8.3 zeigt, aber in der Praxis scheitert die Berechnung bereits für relativ kleine Argumente an der riesigen Rekursionstiefe.

So erfordert bereits die Ausrechnung von $A(1,3)$ "per Hand" folgenden Aufwand:

$$\begin{aligned} A(1,3) &= A(0,A(1,2)) \\ &= A(0,A(0,A(1,1))) \\ &= A(0,A(0,A(0,A(1,0)))) \\ &= A(0,A(0,A(0,A(0,1)))) \\ &= A(0,A(0,A(0,2))) \\ &= A(0,A(0,3)) \\ &= A(0,4) \\ &= 5 \end{aligned}$$

Satz 8.3 *Für alle $m, n \in \mathbb{N}$ terminiert der Aufruf $A(m,n)$ nach endlich vielen Schritten.*

Beweis: Der Beweis erfolgt durch zwei ineinander geschachtelte Induktionen über m (*äußere Induktion*) und, bei festem m, über n (*innere Induktion*).

Induktionsanfang ($m = 0$): Dann erfolgt unabhängig von n nur ein Aufruf. Also wird terminiert.

Induktionsvoraussetzung: Die Behauptung sei richtig für alle k mit $0 \leq k < m$, und für alle n, d. h. der Aufruf $A(k,n)$ terminiert nach endlich vielen Schritten.

Schluß auf m durch Induktion über n (*innere Induktion*):

Induktionsanfang ($n = 0$): Dann wird für $A(m,0)$ der Wert von $A(m-1,1)$ zurückgegeben. Hierfür terminiert der Algorithmus nach Induktionsvorausetzung der äußeren Induktion.

Induktionsvoraussetzung: Der Aufruf $A(m,l)$ terminiert für (festes) m und alle $l < n$.

Schluß auf n: Der Aufruf von $A(m,n)$ erzeugt den Aufruf von $A(m-1, A(m,n-1)$. Nach innerer Induktionsvoraussetzung terminiert der Aufruf $A(m,n-1)$ und liefert eine Zahl k. Dies erzeugt den Aufruf $A(m-1,k)$, der nach äußerer Induktionsvorausetzung terminiert. □

Die Ackermann Funktion wächst sehr stark, und zwar, wie in der *Theorie der rekursiven Funktionen* oder der *Berechenbarkeit* gezeigt wird, stärker als jede sogenannte *primitiv rekursive* Funktion (das sind Funktionen mit einem "einfachen" Rekursionsschema).

Es gilt z. B. (vgl. Übung):

$$
\begin{array}{ll}
A(0,n) > n & A(3,n) > 2^n \\
A(1,n) > n+1 & A(4,n) > 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \quad n \text{ mal} \\
A(2,n) > 2n & A(5,4) > 10^{10000}
\end{array}
$$

8.1.4 Ulams Funktion

Diese Funktion wurde mehrfach von Mathematikern untersucht (Klam, Collatz, Kakutani, vgl. [LV92]). Sie stellt ein Beispiel für einen einfachen Algorithmus dar, für den bis heute nicht bekannt ist, ob er bei allen Eingaben terminiert.

Der Algorithmus erzeugt, ausgehend von einer natürlichen Zahl $a_0 > 0$, eine Folge von Zahlen $a_0, a_1, \ldots, a_n, \ldots$ gemäß der Vorschrift

$$
a_{n+1} = \begin{cases} a_n/2\,, & \text{falls } a_n \text{ gerade ist,} \\ 3a_n + 1\,, & \text{sonst,} \end{cases}
$$

und bricht ab, sobald $a_n = 1$ gilt.

Zum Beispiel führt $a_0 = 3$ zur Folge 3, 10, 5, 16, 8, 4, 2, 1. Es ist offen, ob der Algorithmus bei beliebigen Input stets zum Abbruch führt, d. h. ob die *Länge der Folge* immer endlich ist.

Wir kleiden dies in eine rekursive Funktion. Sei $a \in \mathbb{N}$, a > 0

$$
ulam(a) := \begin{cases} 0\,, & \text{falls } a = 1\,, \\ 1 + ulam(a/2)\,, & \text{falls } a \text{ gerade}, a \geq 2\,, \\ 1 + ulam(3a+1)\,, & \text{falls } a \text{ ungerade}\,, a \geq 2\,. \end{cases}
$$

Offenbar berechnet $ulam(a)$ gerade die Länge der Folge mit $a_0 = a$.

Obwohl wir nicht wissen, ob diese Funktion für jede natürliche Zahl als Input einen Funktionswert liefert, ist die Definition als C++ Funktion natürlich zulässig. Es ist jedoch die (zumindest theoretische) Möglichkeit nicht auszuschließen, daß ein Aufruf der Funktion *ulam* für bestimmte Argumente eine nicht abbrechende Folge von rekursiven Aufrufen in Gang setzt. Dies ist zugleich ein Beispiel für den Fall, daß eine formal zulässige Funktionsdefinition nicht auch inhaltlich vernünftig sein muß. Man sollte sich stets vergewissern, ob der durch eine Funktionsdefinition beschriebene Berechnungsprozeß für beliebige Argumente abbricht, also auch für solche, an die man vielleicht zunächst nicht gedacht hat.

8.2 Wo Rekursion zu vermeiden ist

Rekursive Algorithmen eignen sich besonders, wenn das zugrunde liegende Problem oder die zu behandelnden Daten rekursiv definiert sind. Das bedeutet aber nicht, daß eine solche rekursive Definition eine Garantie dafür bietet, daß ein rekursiver Algorithmus der beste Weg zur Lösung des Problems ist.

Der Aufwand bei rekursiven Aufrufen wird im wesentlichen durch den *Aufrufbaum* (Rekursionsbaum) bestimmt. Seine Höhe plus 1 ist die *Rekursionstiefe*. Sie bestimmt die *maximale*

Größe des Run-Time-Stacks, der durch den ersten Aufruf verursacht wird, und bestimmt damit den benötigten *Speicherplatz*.

Die Gesamtanzahl der Knoten im Aufrufbaum bestimmt die *Anzahl aller rekursiven Aufrufe*. Sie ist ein Maß für die benötigte *Laufzeit*.

Als *Generalregel* sollte man daher Rekursion immer dann vermeiden, wenn der benötigte Speicherplatz (also die Rekursionstiefe) oder die benötigte Laufzeit (also die Anzahl der Knoten des Rekursionsbaums) zu groß werden. Wir erläutern dies an einigen Beispielen.

8.2.1 Berechnung der Fakultät

Die Implementation der Fakultätsfunktion gemäß der rekursiven Definition

$$Fak(n) := \begin{cases} 1\,, & \text{falls } n = 0\,, \\ n \cdot Fak(n-1)\,, & \text{falls } n > 0\,, \end{cases}$$

führt zur Rekursionstiefe $n+1$, die deutlich zu groß ist. Der Rekursionsbaum entartet zudem zu einer Liste (keine Verzweigungen), was meist ein Zeichen dafür ist, daß es auch einen einfachen iterativen Algorithmus gibt (vgl. Beispiel 7.1).

8.2.2 Berechnung des größten gemeinsamen Teilers

Der rekursive Algorithmus in Kapitel 8.1.1 führt zu einer Rekursionstiefe von $1+2\cdot\log\max\{n,m\}$ (Satz 8.1). Dies ist akzeptabel. Allerdings ist der Rekursionsbaum wieder eine Liste, do daß man auch einen iterativen Algorithmus verwenden könnte (vgl. Algorithmus 3.2).

8.2.3 Die Türme von Hanoi

Die Rekursionstiefe ist bei n Scheiben $n+1$, und die Gesamtanzahl der Aufrufe ist $2^{n+1}-1$. Dies ist sehr groß, so daß eine iterative Lösung vorzuziehen wäre, wenn sie denn einfach zu finden wäre. Die rekursive Struktur des Problems legt jedoch die Verwendung der Rekursion nahe.

8.2.4 Berechnung der Fibonacci-Zahlen

Die Folge $f_0, f_1, f_2, \ldots$ der *Fibonacci-Zahlen* wächst nach dem Gesetz

$$\begin{aligned} &f_0 := 0, \quad f_1 := 1, \\ &f_{n+1} := f_n + f_{n-1} \text{ für } n > 0\,. \end{aligned}$$

Sie stellen eine schnell wachsende Folge von Zahlen dar, die das Wachstum einer Population von sich schnell vermehrenden Lebewesen (Bakterien, Kaninchen) modelliert. Ist f_n die Anzahl der "Neugeburten" in Periode n, und reproduzieren sich in einer Periode genau die in den beiden vorherigen Perioden "geborenen" Mitglieder ("gebärfähiges Alter") so entsteht die Folge der Fibonacci-Zahlen als Folge der Geburtenzahlen.

Die rekursive Definition führt auf folgende rekursive Funktion:

```
int Fib( /* in */ int n)
{
    if ( n == 0 ) return 0;
    else if ( n == 1 ) return 1;
    else return Fib(n-1) + Fib(n-2);
}
```

Beim Aufruf von `Fib(5)` ergibt sich der in Abbildung 8.10 dargestellte Rekursionsbaum.

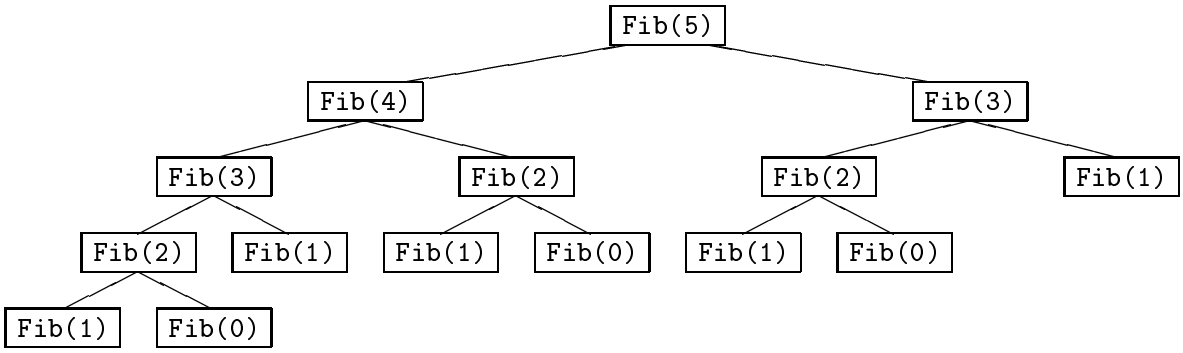

Abbildung 8.10: Rekursionsbaum zu der Fibonaccifolge.

Beim Aufruf von `Fib(n)` ist also die Rekursionstiefe n, und für die Gesamtanzahl $T_{Fib}(n)$ der rekursiven Aufrufe gilt

Satz 8.4 *$T_{Fib}(n)$ wächst mindestens exponentiell. Genauer gilt*

$$T_{Fib}(n) \geq 2^{\lfloor n/2 \rfloor} \text{ für } n \geq 2\,.$$

Beweis: Dies beweist man durch vollständige Induktion wie folgt:

Induktionsanfang:

$$\begin{aligned} n = 0: &\ T_{Fib}(0) = 1,\ 2^{\lfloor 0/2 \rfloor} = 2^0 = 1 \\ n = 1: &\ T_{Fib}(1) = 1,\ 2^{\lfloor 1/2 \rfloor} = 2^0 = 1 \end{aligned}$$

Schluß von n auf $n+1$:

$$\begin{aligned} T_{Fib}(n+1) &\geq 2 \cdot T_{Fib}(n-1) \\ &\geq 2 \cdot 2^{\lfloor (n-1)/2 \rfloor} \\ &= 2^{\lfloor (n+1)/2 \rfloor}\,. \end{aligned}$$

Die erste Ungleichung folgt aus der Tatsache, daß $Fib(n-1)$ zweimal aufgerufen wird; die zweite folgt aus der Induktionsvoraussetzung. □

Der Aufwand ist also ähnlich wie bei den Türmen von Hanoi. Während jedoch bei den Türmen von Hanoi stets andere Teilprobleme in den rekursiven Aufrufen berechnet werden, tauchen hier *dieselben Teilprobleme* wiederholt auf.

Eine solche *Mehrfachberechnung identischer Teilprobleme* sollte man unbedingt vermeiden. Bei den Fibonacci-Zahlen geht dies durch folgende iterative Variante.

Der letzte `else` Teil in `Fib` wird dabei ersetzt durch

```
{
    int currentFib = 1, prevFib = 0;
    for (int i = 1, i < n; i++){
        currentFib = currentFib + prevFib;
        prevFib    = currentFib - prevFib;
    }\\endfor
}
```

Dabei spielen die Variablen `currentFib` und `prevFib` die Rolle von f_n und f_{n-1}, die in beiden Zuweisungen zu $f_{n+1} = f_n + f_{n-1}$ und $f_n = f_{n+1} - f_{n-1}$ aktualisiert werden.

Man vergleiche einmal die Laufzeit beider Varianten für Zahlen n ab $n = 35$.

8.2.5 Zusammenfassung

Die Folgerung aus diesen Überlegungen ist, daß man auf die Verwendung von Rekursion immer dann verzichten sollte, wenn es eine offensichtliche Lösung mit Iteration gibt. Das bedeutet aber nicht, daß Rekursion um jeden Preis zu umgehen ist. Es gibt viele gute Anwendungen für Rekursion, wie die folgenden Kapitel zeigen werden.

Die Tatsache, daß Implementationen von rekursiven Funktionen auf nicht-rekursiven Maschinen existieren, beweist, daß gegebenenfalls jedes rekursive Programm in ein rein iteratives umgeformt werden kann. Dies verlangt jedoch das explizite Verwalten eines Rekursions-Stacks. Durch diese Operationen wird das Grundprinzip eines Programms oft so sehr verschleiert, daß es schwer zu verstehen ist. Zusammenfassend läßt sich sagen, daß Algorithmen, die ihrem Wesen nach eher rekursiv als iterativ sind, tatsächlich als rekursive Funktionen formuliert werden sollten.

8.3 Literaturhinweise

Die hier gegebene Darstellung lehnt sich in Teilen an [Wir86] an. Von dort ist auch Abbildung 8.1 entnommen.

Die Darstellung der Türme von Hanoi folgt [OW82].

Der Artikel [LV92] gibt vertiefende Informationen (und Berechnungen) zu Ulams Funktion.

Weitere Beispiele für Rekursion finden sich in nahezu allen Büchern über Algorithmen und Datenstrukturen.