SCHRIFTLICHE HAUSARBEIT

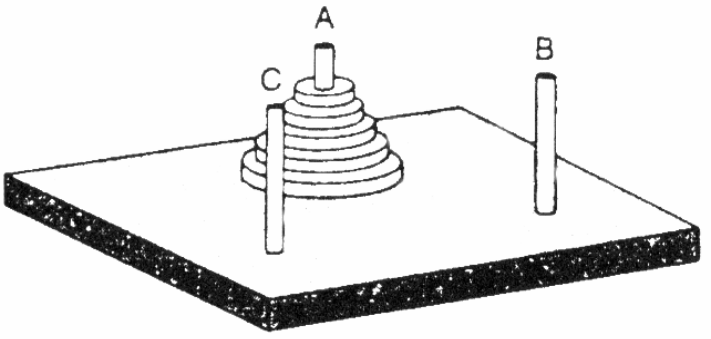
zur Abschlussprüfung der erweiternden Studien für Lehrer

im Fach Informatik

VIII. Weiterbildungskurs in Zusammenarbeit mit der Fernuniversität Hagen

Eingereicht dem Amt für Lehrerausbildung Fuldatal

Das Thema „Rekursion“ im Informatikunterricht



Verfasserin: Corina Kreß

Gutachter: StD Otto Wehrheim

INHALTSVERZEICHNIS

VORWORT

Die vorliegende Arbeit beschäftigt sich mit einem wichtigen algorithmischen Prinzip zur Problemlösung, der Rekursion.

In Kapitel 1 wird zunächst an einigen alltäglichen Beispielen, die den meisten von uns schon einmal begegnet sind, die Rekursion mit ihren wesentlichen Eigenschaften erläutert. Anschließend wird anhand einiger Beispiele aus der Mathematik das Prinzip auf einer höheren Abstraktionsebene verdeutlicht. Was Rekursion in der Informatik bedeutet, wird anschließend dargestellt.

Kapitel 2 widmet sich der Umsetzung des Themas im schulischen Informatikunterricht. Dabei wird zunächst darauf eingegangen, an welchen Stellen im Informatikunterricht das Thema „Rekursion“ eine Rolle spielt. Anschließend wird auf methodische Fragen eingegangen.

In Kapitel 3 wird das Konzept der Rekursion mit dem der Iteration verglichen. Jedes der beiden hat bestimmte Vor- und Nachteile, die an dieser Stelle anhand von Beispielen erläutert werden.

Die Standard-Probleme in der Informatik, die typischerweise mit Rekursion gelöst werden, dürfen natürlich in dieser Arbeit nicht fehlen und werden in Kapitel 4 vorgestellt.

Im Anhang finden sich zahlreiche Arbeitsblätter für den Unterricht, die die Themen dieser Arbeit aufgreifen und noch darüber hinausgehen. Teilweise sind auch die Lösungen dazu angegeben. An dieser Stelle möchte ich mich herzlich bei Herrn Daniel Garmann (Gymnasium Odenthal) bedanken, der mir freundlicherweise seine Arbeitsblätter zur Verfügung gestellt hat, auf die ich im Zuge meiner Internet-Recherche gestoßen bin.

Die in der Arbeit vorgestellten Programme sind in der Programmiersprache Pascal bzw. Delphi verfasst, da meines Erachtens die Programme in dieser Sprache durch ihren einfachen, selbstredenden Aufbau leicht verständlich sind und daher gut geeignet, das Programmierprinzip zu verdeutlichen.

Herzlich danken möchte meinem Betreuer Otto Wehrheim für die gute Betreuung und die wertvolle Unterstützung mit Materialien zum Thema.

Frankfurt, im September 2004

1 GRUNDLAGEN

Um Rekursion zu verstehen, muss man entweder einen kennen, der sie versteht, oder sie schon verstanden haben.
M. Freericks

Was ist Rekursion?

Rekursion bedeutet Selbstbezüglichkeit (von lateinisch *recurrere* = zurücklaufen). Sie tritt immer dann auf, wenn etwas auf sich selbst verweist.

Ein Objekt wird auch rekursiv genannt, wenn „es sich selbst als Teil enthält oder mithilfe von sich selbst definiert ist.“ [1]

Eine andere Definition spricht von Verschachtelung und Varianten der Verschachtelung.[2]

1.1 Beispiele aus dem Alltag

In der Literatur werden vielerlei Beispiele zur Veranschaulichung der Rekursion genannt. Darunter sind welche, bei denen die Komponente „Zeit“ keine Rolle spielt, d. h. der Selbstbezug auf den ersten Blick erkennbar ist. Es handelt sich hierbei etwa um Verpackungen, auf deren Etikett die gleiche Verpackung dargestellt ist, oder um ein Fernsehbild, das einen Ansager zeigt, neben der ein Fernseher steht, auf der man das Fernsehbild und damit den Ansager wieder sehen kann, usw. Auch ein Gegenstand, der zwischen zwei parallele Spiegel gehalten wird, erscheint vervielfacht.[3]

Abb. 1: Rekursion im Fernsehbild

[1] Wirth 149.

[2] Siehe Hofstadter 137.

[3] Siehe Baumann (1993) 11.

Bezieht man die Zeitkomponente mit ein, so werden als Beispiele Filme innerhalb von Filmen (etwa in Form von Rückblenden) oder Geschichten innerhalb von Geschichten genannt.

Zur Veranschaulichung sei hier ein Beispiel einer Geschichte in einer Geschichte, die wiederum eine Geschichte enthält (usw.), vorgestellt[4]:

> *Es war einmal ein Mann, der hatte 7 Kinder, und die Kinder sprachen: „Vater, erzähl uns eine Geschichte". Und der Vater begann: „Es war einmal ein Mann, der hatte 7 Kinder, und die Kinder sprachen: „Vater, erzähl uns eine Geschichte". Und der Vater begann: „Es war einmal ein Mann, der hatte 7 Kinder, und die Kinder sprachen: ...*

Diese Geschichte kann hier natürlich nicht fertig abgedruckt werden, da sie kein Ende findet. Sie ist sozusagen durch sich selbst definiert. Dies erscheint merkwürdig, erfährt man doch nie, wie die eigentliche Geschichte endet.

Ebenso verhält es sich mit dem bekannten Lied „Ein Mops kam in die Küche":

> *„Ein Mops kam in die Küche*
> *und stahl dem Koch ein Ei.*
> *Da nahm der Koch den Löffel*
> *und schlug den Mops zu Brei.*
> *Da kamen viele Möpse*
> *und gruben ihm ein Grab*
> *und setzten einen Grabstein,*
> *auf dem geschrieben stand:*
> > *„Ein Mops kam in die Küche*
> > *und stahl dem Koch ein Ei.*
> > *Da nahm der Koch den Löffel*
> > *und schlug den Mops zu Brei.*
> > *Da kamen viele Möpse*
> > *und gruben ihm ein Grab*
> > *und setzten einen Grabstein,*
> > *auf dem geschrieben stand:*
> > > *„Ein Mops kam in die Küche*
> > > *(usw.)*

Alle bisher genannten Beispiele sind zwar Beispiele für Rekursionen, da jeweils ein Selbstbezug bzw. eine -enthaltung auftritt, und sie dienen daher der Veranschaulichung von Rekursion; doch sie verfolgen keinen Zweck. Sie rufen lediglich beim Betrachter/Zuhörer Interesse hervor bzw. haben eine einprägende Wirkung, z. B. bei der Werbung. Sie sind zu oberflächlich, um wiedergeben zu können, welches leistungsfähige Werkzeug die Rekursion in der Informatik ist. Weitere Ausführungen hierzu folgen in Kapitel 1.3.

[4] Kuhlmann 240.

Neben diesen Beispielen, die uns hier und da im täglichen Leben begegnen, tritt die Rekursion auch in *Handlungen* des täglichen Lebens auf.

Im Folgenden wird ein Beispiel dafür vorgestellt:

> *Bei modernen Telefonen ist es möglich, mehrere Telefonate gleichzeitig zu empfangen. Während man mit Person A spricht, kann ein weiterer Anrufer B „anklopfen". Durch einen entsprechenden Knopfdruck kann man nun A in eine Warteposition bringen und mit B telefonieren. Meldet sich ein weiterer Anrufer C, so kann auch B in die Wartestellung versetzt werden, damit das Telefonat mit C geführt werden kann. Wird das Gespräch mit C beendet, kann das Gespräch mit B weitergeführt werden. Ist dieses wiederum beendet, kehrt man zum Gespräch mit A zurück. Allerdings könnte es auch vorkommen, dass sich ein neuer Anrufer D meldet, nachdem das Gespräch mit B wieder aufgenommen wurde. B wird dann noch einmal in die Warteposition versetzt, das Gespräch mit D erledigt, und dann das mit B zu Ende geführt. Schließlich kehrt man nach A zurück.*

Ein weiteres Beispiel aus unserem Alltag ist folgendes:

> *Bei Nachrichtensendungen im Radio oder Fernsehen wird oft zu einem auswärtigen Korrespondenten umgeschaltet. Dieser Korrespondent wiederum spielt beispielsweise ein Tonband oder einen Filmausschnitt von einem Lokalreporter, der jemanden vor Ort interviewt hat, ab, nachdem er etwas über die Hintergründe gesagt hat. Danach spricht der Korrespondent noch ein paar abschließende Worte und gibt zurück ins Radio- oder Fernsehstudio.* [5]

Häufig tritt Rekursion dann in Handlungen auf, wenn die Erfüllung einer Aufgabe zugunsten einer einfacheren Aufgabe, die von derselben Art ist, aufgeschoben wird.

Auch in der Musik tritt Rekursion auf.[6] Ein Beispiel dafür ist die Sonatenhauptsatzform in ihrer typischen Dreiteiligkeit Exposition, Durchführung und Reprise. In der Exposition werden zwei Themen vorgestellt, das erste in der Grundtonart, das zweite in einer nächstverwandten Tonart. Dadurch wird eine Spannung erzeugt, die in der Durchführung mit ihren stark modulierenden und variierenden Teilen verstärkt wird. Das Ende löst die beim Hörer entstandene Spannung, indem es die beiden Themen der Exposition schließlich in der Grundtonart wieder aufnimmt (=Reprise).

Nicht zuletzt möchte ich auf Rekursion in der Sprache eingehen: Hier haben wir die Möglichkeit, Nebensätze in Sätze einzuschieben, die möglicherweise ihrerseits wieder eingeschobene Nebensätze enthalten. Auch Kommentare in Klammern innerhalb von Kommentaren in Klammern usw. sind ein Beispiel für Rekursion.

---

[5] Beispiele aus Hofstadter 137f.

[6] Nähere Ausführungen hierzu bei Hofstadter 140f.

Ein schönes Beispiel eines verschachtelten Satzes, den ich hier in leicht abgewandelter Form wiedergebe, stellt HOFSTADTER vor: [7]

> *Die notorische Eigenheit der deutschen Sprache, das Verbum ans Ende des Satzes zu stellen, über welche lustige Geschichten von geistesabwesenden Professoren, die einen Satz beginnen, die ganze Vorlesung lang weiterreden, und damit aufhören, dass sie eine Kette von Verben herunterleiern, wobei die Zuhörer, für die die einzelnen Satzbruchteile schon längst jeglichen Zusammenhang verloren haben, völlig verwirrt werden, erzählt werden, ist ein sehr gutes Beispiel für linguistische Rekursion.*

Man könnte die Rekursivität natürlicher Sprachen als Synonym für deren kreative Potenz betrachten: „Wir können durch Anfügen von ‚und'-Sätzen, Verschachtelung von Relativsätzen, Anneinanderreihung von Adjektiven etc. Sätze bilden, die beliebig lang sind. Daß uns dabei das Gedächtnis im Stich läßt, daß uns die Zeit, der Atem oder das Leben ausgehen wird, ist kein Defekt unserer Sprachfähigkeit"[8].

Ein Beispiel aus der Literatur gibt die Einleitung zu Christian Morgensterns *Galgenliedern*[9]:

> *„Es darf daher getrost, was auch von allen, deren Sinne, weil sie unter Sternen, die, wie der Dichter sagt: ‚dörren, statt zu leuchten', geboren sind, vertrocknet sind, behauptet wird, enthauptet werden, daß hier einem sozumaßen und im Sinne der Zeit, dieselbe im Negativen als Hydra gesehen, hydratherapeutischen Moment ersten Ranges, immer angesichts dessen, daß, wie oben, keine mit Rosenfingern den springenden Punkt ihrer schlechthin unvoreingenommenen Hoffnung auf eine, sagen wir, schwansinnige oder wesentielle Erweiterung des natürlichen Stoffgebietes zusamt mit der Freiheit des Individuums vor dem Gesetz Ihrer Volksseele zu verraten sich zu entbrechen den Mut, was sage ich, die Verruchtheit haben wird, einem Moment, wie ihm in Handel, Wandel, Kunst und Wissenschaft allüberall dieselbe Erscheinung, dieselbe Frequenz den Arm bieten, und welches bei allem, ja vielleicht gerade trotz allem, als ein mehr oder minder modulationsfähiger Ausdruck einer ganz bestimmten und im weitesten Verfolge excösen Weltauffasserraumwortkindundkunstanschauung kaum mehr zu unterschlagen versucht werden zu wollen vermag - gegenübergestanden und beigewohnt werden zu dürfen gelten lassen zu müssen sein möchte."*

Komplizierte Konstrukte wie diese kommen im gewöhnlichen Umgangsdeutsch zwar nicht vor, doch stellen sie ein häufig genutztes Stilmittel in der Literatur dar. Ein Beispiel dafür, die Rekursion an Beispielen aus der Liguistik aufzuziehen, findet sich bei ROBERTS[10].

---

[7] Vgl. Hofstadter 141.

[8] Grewendorf 32.

[9] Morgenstern 21.

[10] Roberts 65ff.

1.2 Rekursion in der Mathematik

Besonders in der Mathematik findet die Methode der Rekursion Anwendung. „Die Rekursion (verbunden mit der vollständigen Induktion) ist aus den Grundlagen der Mathematik nicht wegzudenken. Wenn man etwa, auf den Peanoschen Axiomen aufbauend, die Addition und die Multiplikation von natürlichen Zahlen definieren will, kommt nur eine rekursive Definition in Betracht.“[11] Diese ist in Abschnitt 1.2.1 dargestellt.

Im Folgenden werden außerdem einige weitere mathematische Definitionen in Form von Funktionen vorgestellt, in denen die Rekursion zur Geltung kommt.

„Die Grundidee der rekursiven Definition einer Funktion *f* ist: Der Funktionswert *f*(n+1) einer Funktion *f*: $\mathbf{N_0} \rightarrow \mathbf{N_0}$ ergibt sich durch Verknüpfung bereits vorher berechneter Werte *f*(n), *f*(n-1), ... Falls außerdem die Funktionswerte von *f* für hinreichend viele Startargumente bekannt sind, kann jeder Funktionswert von *f* berechnet werden. Das heißt im Klartext: Bei einer rekursiven Definition einer Funktion *f* ruft sich die Funktion so oft selber auf, bis ein vorgegebenes Argument (meistens 0) erreicht ist, so dass die Funktion terminiert.“[12]

Die Rekursion erscheint hier als eine Methode, um bestimmte, meist komplizierte, Funktionen einfacher oder überhaupt darstellen und/oder lösen zu können. Dabei ist wesentlich, dass das Problem in einzelne, voneinander abhängige Teile zerlegt wird, und somit Schritt für Schritt gelöst werden kann.

1.2.1 Die Grundrechenarten[13]

Zunächst sei hier die Nachfolgerfunktion *S* definiert: $S: \mathbf{N_0} \rightarrow \mathbf{N_0}$ mit $S(n) = n+1$.

Addition: $add(0, m) = 0 + m = m;$

$$add(n, m) = n + m = ((n\text{-}1)+m)+1 = S(add(n\text{-}1, m))$$

Die Summe aus *n* und *m* ist der Nachfolger der Summe aus *n-1* und *m*.

[11] Hafenbrak 108.

[12] http://de.wikipedia.org/wiki/Rekursion.

[13] Definitionen aus Engelmann 440.

Multiplikation: $mult(0, m) = 0 \cdot m = 0;$

$mult(n, m) = n \cdot m = (n\text{-}1) \cdot m + m = add(mult(n\text{-}1, m), m)$

Das Produkt aus n und m ist die Summe aus m und dem Produkt aus $n\text{-}1$ und m.

Auch das Potenzieren kann rekursiv definiert werden:

$pot(0, b) = b^0 = 1;$

$pot(n, b) = b^n = b^{n-1} \cdot b = mult(pot(n\text{-}1, b), b)$

Die Potenz b^n ist das Produkt aus der Potenz b^{n-1} und b.

1.2.2 Die Summe der ersten *n* natürlichen Zahlen

Die rekursive Definition der Summe der ersten n natürlichen Zahlen

$sum(n) = 0 + 1 + 2 + \ldots + n$ lautet:

$sum(0) = 0$ (Rekursionsanfang)

$sum(n) = sum(n\text{-}1) + n$ (Rekursionsschritt)

Die Summe der ersten n Zahlen lässt sich also berechnen, indem man die Summe der ersten n-1 Zahlen berechnet und dazu die Zahl n addiert. Damit die Funktion terminiert, legt man hier $sum(0) = 0$ (Rekursionsanfang) fest. Mit diesen Angaben lässt sich eine rekursive Definition angeben.

So gilt zum Beispiel:

$sum(4)$	$= sum(3) + 4$	(Rekursionsschritt)
	$= sum(2) + 3 + 4$	(Rekursionsschritt)
	$= sum(1) + 2 + 3 + 4$	(Rekursionsschritt)
	$= sum(0) + 1 + 2 + 3 + 4$	(Rekursionsschritt)
	$= 0 + 1 + 2 + 3 + 4$	(Rekursionsanfang)
	$= 10$	

1.2.3 Die Fakultäts-Funktion

Die Fakultäts-Funktion $f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n$ ist ebenfalls rekursiv definiert (vgl. Definition der Summe der ersten n natürlichen Zahlen im letzten Abschnitt):

$f(0) = 0! = 1;$

$f(n) = n! = f(n\text{-}1) \cdot n$

Diese Funktion gibt die Anzahl der Permutationen von n verschiedenen Elementen an.

Die rekursive Definition der Fakultäts-Funktion findet man in vielen Lehrbüchern als Einstiegsbeispiel in das Thema Rekursion. In den Abschnitten 2.2 und 3.1 werde ich noch näher darauf eingehen.

1.2.4 Die Fibonacci-Folge

Die Fibonacci-Folge $fib(n)$ von Zahlen ist rekursiv definiert:

$fib(0) = 0;\ fib(1) = 1;$

$fib(n) = fib(n\text{-}1) + fib(n\text{-}2)$ für $n > 1$ [14]

In diesem Beispiel erscheinen erstmals zwei Aufrufe im Funktionskörper.

Die Fibonacci-Folge hat u. a. eine große Bedeutung in der Natur (Kaninchenvermehrung, Sonnenblume, Blattwinkel, …). Die vielen interessanten Facetten der Folge sind in folgender Grafik dargestellt:

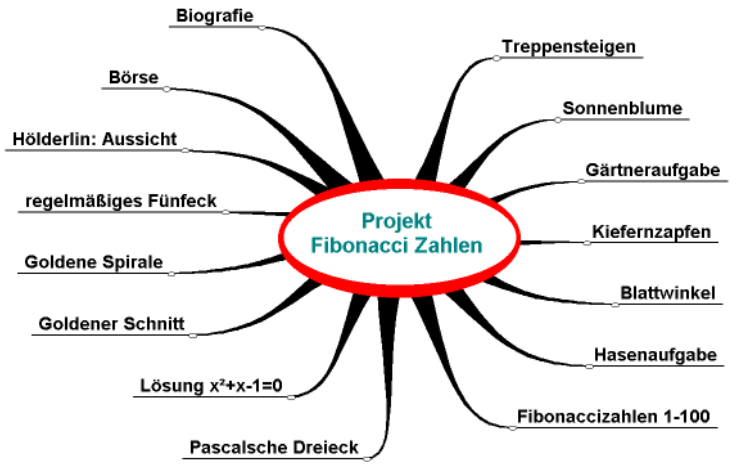


Abb. 2: Facetten der Fibonacci-Folge[15]

[14] Im Anhang befindet sich ein Arbeitsblatt zur iterativen und rekursiven Entwicklung der Fibonacci-Zahlen sowie die entsprechenden Delphi-Programme.

[15] Abb. aus http://www.mathekiste.de/fibonacci/projfibonacci.htm; Erläuterungen dazu finden sich ebenfalls auf diesen Seiten.

1.2.5 Das Rad des Theodorus[16]

Das Rad des Theodorus (griechischer Gelehrter, 465 v. Chr.) ist eines der ersten Beispiele einer Rekursion:[17]

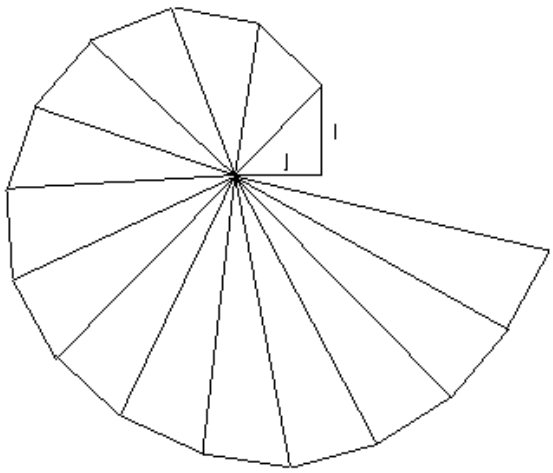


Abb. 3: Das Rad des Theodorus[18]

Es kann durch einen rekursiven Algorithmus nach folgender Regel gebildet werden:

D_1 ist ein rechtwinkliges, gleichschenkliges Dreieck mit Schenkellänge 1.
D_n ist ein rechtwinkliges Dreieck,
wobei ein Schenkel gleich der Hypotenuse von D_{n-1} ist
und der andere Schenkel die Länge 1 hat.

Die Länge der Hypotenuse c_n von D_n ist somit auch rekursiv definiert und kann nun mathematisch notiert werden:

$$c_1 = \sqrt{2}$$
$$c_n = \sqrt{{c_{n-1}}^2 + 1}$$

[16] Beispiel genannt in: http://www.fim.uni-linz.ac.at/lva/rus/Rekursion/Rekursion.htm.

[17] Mit Hilfe dieser Konstruktion soll Theodorus erstmals bewiesen haben, dass $\sqrt{3}, \sqrt{5}, \sqrt{7} \ldots \sqrt{17}$ irrationale Zahlen sind.

[18] Abb. aus http://www.fim.uni-linz.ac.at/lva/rus/Rekursion/Rekursion.htm.

1.2.6 Die Binomialkoeffizienten / Das Pascal'sche Dreieck[19]

Das Pascal'sche Dreieck ist ein Hilfsmittel, das die Ermittlung der Binomialkoeffizienten auch demjenigen möglich macht, der mit der Bildung von $\binom{n}{k}$ nicht vertraut ist.[20] Man erhält es, indem man, beginnend mit $(a + b)^0 = 1$ und $(a + b)^1 = a + b$, die Koeffizienten des Terms auf der rechten Seite dreieckförmig untereinander schreibt.

Dabei entstehen die Zahlen jeder Zeile, indem die zwei benachbarten Zahlen der darüber stehenden addiert werden. Das Randfeld erhält jeweils den Wert 1.

Das Pascal'sche Dreieck ist ein Beispiel für Rekursion, bei dem die rekursive Methode sofort zu erkennen ist.

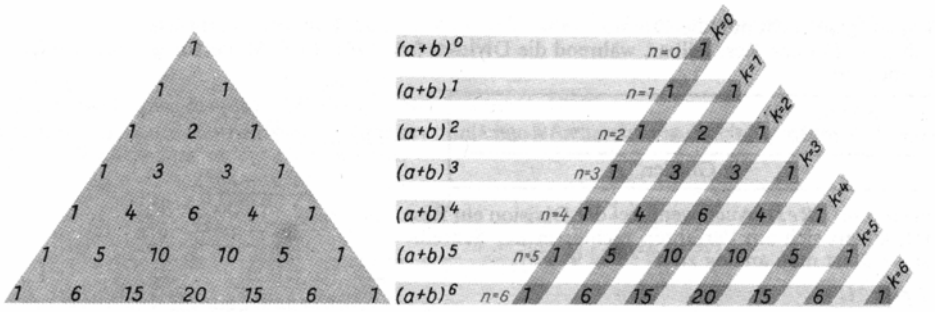


Abb. 4: Das Pascal'sche Dreieck[21]

In mathematischer Notation ist das Pascal'sche Dreieck folgendermaßen definiert:

$$\binom{n}{0} = 1; \quad \binom{n}{n} = 1; \quad \binom{n}{k} = 0 \text{ für } k > n; \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \text{bzw.}$$

$$f(n,0) = 1;\ f(n,n) = 1;\ f(n,k) = 0 \text{ für } k > n;\ \ f(n, k) = f(n\text{-}1, k\text{-}1) + f(n\text{-}1, k)\ ,$$

wobei n die aktuelle Zeile und k die aktuelle Spalte angibt.

Die Mathematik bietet noch viele andere rekursive Definitionen und die Informatik eine Fülle von Problemen, die durch rekursive Überlegungen elegant gelöst werden können.

---

[19] Im Anhang: Arbeitsblatt und entsprechendes Delphi-Programm.

[20] Zur Erinnerung: Der Binomische Lehrsatz lautet:

$$(a+b)^n = \sum_{k=o}^{n} \binom{n}{k} a^{n-k} b^k = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \ldots + \binom{n}{n-1} a b^{n-1} + \binom{n}{n} b^n$$

[21] Abb. aus Gellert 35.

1.3 Rekursion in der Informatik

1.3.1 Rekursion als Problemlösewerkzeug

Die Rekursion in den bisher aufgeführten Beispielen kann belustigen und erstaunen, dient in Musik und Literatur als Stilmittel oder ist in der Mathematik ein Weg, Funktionen zu definieren. In der Informatik dient das Prinzip der Rekursion vor allem dazu, **Probleme** möglichst elegant **zu lösen**.

In der Informatik ist die Rekursion ein sehr leistungsfähiges **Problemlöse-Werkzeug**.

Dies gelingt dadurch, dass bei jedem erneuten Aufruf einer Prozedur/Funktion von sich selbst eine Vereinfachung eintritt, die letztlich nach endlich vielen Aufrufen zum Ziel führt. Bei den Beispielen aus der Mathematik ist dies ebenfalls gegeben, bei den genannten Alltagsbeispielen (TV, Etikett usw.) allerdings nicht.

Insofern ist die Beschreibung der Rekursion als die „Definition eines Problems, einer Funktion oder eines Verfahrens durch sich selbst“ (Duden Informatik) zunächst nicht ganz korrekt. Denn eine korrekt formulierte rekursive Definition definiert „nie etwas durch sich selbst, sondern immer durch *einfachere Versionen* ihrer selbst“[22]. Und auch der Duden präzisiert seine Definition noch: „Das Ergebnis eines Verfahrens für die Eingabe x wird auf das Ergebnis des gleichen Verfahrens für eine Eingabe y zurückgeführt (ist hierbei $x = y$, dann lässt sich kein Ergebnis ermitteln). Ebenso wird das Ergebnis bzgl. y auf das Ergebnis des Verfahrens bzgl. z zurückgeführt usw.“

Ein Algorithmus ist rekursiv, wenn er mit Hilfe einfacher Varianten von sich selbst beschrieben wird.[23]

Weiter heißt es im Informatik-Duden: „Damit dieses Vorgehen endet, muss mindestens für einen Wert das Ergebnis des Verfahrens bekannt sein.“[24]

---

[22] Hofstadter 137.

[23] Siehe Wehrheim 6.

[24] Duden Informatik 600.

Mit anderen Worten: Ein rekursiver Algorithmus muss eine Abbruchbedingung enthalten. Das ist leicht einzusehen, denn wenn ein Programm ein Problem lösen soll, muss es zu einem Ergebnis kommen, d. h. es muss terminieren und darf sich nicht in einer Endlosrekursion verlieren.

Aus diesem Grund sind die in den meisten Büchern genannten Beispiele zur Rekursion sowie auch die oben beschriebenen nur sehr eingeschränkt geeignet zur Veranschaulichung der Rekursion in der Informatik. Zwar verdeutlichen sie die Rekursion in dem Sinne, als dass sie sich selbst inmitten ihres eigenen Geschehnisses aufrufen, dies geschieht jedoch ohne eine Vereinfachung ihrer Selbst. So kommt es zwangsläufig zu einem unendlichen Prozess; in einem Programm würde dies zur Endlosschleife führen. Zwar bricht bei den genannten Verpackungsetiketten oder dem Fernsehbild das wiederholte Darstellen früher oder später ab, da ein weiteres Bild im Bild irgendwann aus Platzgründen nicht mehr darstellbar ist. Es gibt jedoch keine logische Abbruchbedingung, die von vornherein vorgesehen ist.

Jeder rekursive Algorithmus und jede rekursive Prozedur muss eine Abbruchbedingung enthalten.[25]

Betrachten wir das Beispiel der Russischen Puppen (Matroschkas): Hier tritt Rekursion auf, da jede Puppe eine kleinere Puppe ihrer Art enthält – bis auf die letzte, die kleinste Puppe. Im Gegensatz zu den bisher genannten Alltagsbeispielen sind hier die verschiedenen Stufen der Rekursion zu erkennen: Die Russischen Puppen *vereinfachen* sich insofern, als dass jede kleinere Puppe weniger Puppen beinhaltet als die vorangegangene größere. Die Reihe lässt sich so lange fortsetzen, bis man auf die kleinste Puppe stößt, was zum Abbruch führt.

Zu beachten ist jedoch, dass es auch bei den Matroschkas nicht um das Lösen eines Problems geht.

[25] Siehe Wehrheim 6.

Im Folgenden wird nun ein Beispiel vorgestellt, das den Problemlösecharakter der Rekursion, die notwendigen Vereinfachungen sowie die Abbruchbedingung treffend veranschaulicht:

> *Die Jugendzeit war in früheren Jahren für die Kinder, insbesondere für Jungen, in der Regel eine harte Zeit: Wenig Spielzeug, kaum Fernseher, keinen Computer und häufig lautete abends, vor dem Waschen, der Auftrag der Mutter: "Geh erst einmal in den Keller und hole solange Kohlen herauf, bis die Kohlenkästen der Öfen wieder gefüllt sind!".*
>
> *Wohl dem Jungen, der noch Brüder hatte. Denn dann bestand die Möglichkeit, diese mehrfache und mühselige Schlepperei zum Teil zu delegieren: "Ich hole einen Eimer Kohle. Solange die Kohlenkästen nicht voll sind, gehst du in den Keller und holst die restlichen Kohlen". Die Brüder des Jungen kommen natürlich auch nicht so einfach davon. Denn auch sie erhalten den Auftrag: "Solange die Kohlenkästen nicht voll sind, geh in den Keller und hole einen Eimer Kohle!".*
>
> *Der dritte Sohn freut sich vielleicht, wenn er feststellt, dass bereits alle Kohlenkästen voll sind, und meldet seinem Bruder: "Fertig!", dieser wiederum seinem Bruder: "Fertig!", dieser wiederum der Mutter: "Alles fertig!".* [26]

An dieser Geschichte wird deutlich, dass sich das Problem mit jedem neuen Auftrag „Kohle holen“ vereinfacht: Die Kohlekästen werden von Mal zu Mal voller, und der gerade beauftragte Junge muss für das Heranschaffen einer geringeren Menge Kohle sorgen als noch sein Vorgänger. Dies führt schließlich zu der obligatorischen Abbruchbedingung: Irgendwann sind die Kästen voll, und der zuletzt beauftragte Junge kann seine Aufgabe abschließen und sein „Erledigt!“ zurückmelden, was den Jungen, der zuvor einen Eimer geholt hat, dazu veranlasst, seine Aufgabe ebenfalls als erfüllt zurückzumelden usw.

Letztlich holt jeder Junge genau einen Eimer Kohle, da er jeweils die restliche Arbeit an einen seiner Brüder delegiert (vorausgesetzt, die Familie hat genügend Söhne).

Das Prinzip des Delegierens ist ein wichtiges und grundlegendes Schema einer rekursiven Problemlösung.[27]

Jeder Junge geht folgendermaßen vor:

- Er führt den Teilalgorithmus *Einen Eimer Kohle holen* einmal aus.
- Dann delegiert er den Auftrag *Kohlen holen* an seinen Bruder.
- Dieser delegiert den Auftrag solange weiter an einen seiner Brüder, bis die Abbruchbedingung erfüllt ist, nämlich dass alle Kohlenkästen voll sind.

[26] Geschichte aus Damann, 221 ff.

[27] Vgl. Damann 221.

Der rekursive Algorithmus dazu ist dann auch erstaunlich einfach:

Rekursiver Algorithmus: *Kohlen holen*

```
wenn Kohlenkästen nicht voll,
      dann: Hole einen Eimer Kohle
            Auftrag (an den Bruder): Kohlen holen
```

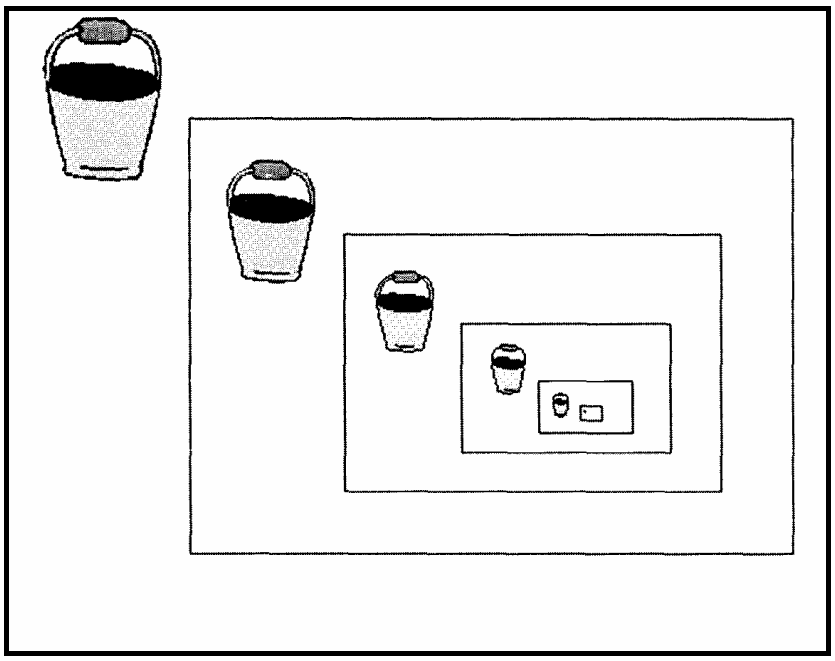


Abb. 5: Verschachtelungen zum Beispiel „Kohlen holen“[28]

Durch die „Erledigt“-Rückmeldung desjenigen Jungen, der mit seinem Eimer Kohle schließlich die Kohlekästen voll macht, wird die Rückführung in Gang gesetzt. So kann zunächst derjenige, der zuvor einen Eimer geholt hat, seine Teilaufgabe ebenfalls abschließen usw. Schließlich wird der von der Mutter beauftragte Sohn zurückmelden, dass die Gesamtaufgabe erledigt ist. Diese Rückführung ist zwingend, damit letztendlich das eigentliche, meist komplizierte Problem als gelöst gemeldet werden kann.

Bei der Entwicklung eines rekursiven Lösungsverfahrens müssen generell folgende drei Punkte geklärt werden:

- rekursiver Abstieg: Wie lautet eine einfachere Version des gegebenen Problems?
  Hier ist dies die Delegierung des Auftrags „Kohle holen“ an den jüngeren Bruder.
- Rekursionsanfang: Wodurch ist der direkt lösbare Fall gekennzeichnet?
  Dieser ist hier dadurch gegeben, dass der letzte Junge in der Reihe die Kohlenkästen füllt.

[28] Abb. aus Damann 222.

- rekursiver Abstieg: Wie gewinnen wir aus der Lösung der einfacheren Problemversion die Lösung des ursprünglichen Problems?
  Dies geschieht in diesem Beispiel durch die „Erledigt!“-Rückmeldung.[29]

Auf der Programmierebene wird dies folgendermaßen ausgedrückt:

„Eine rekursive Prozedur (bzw. Funktion) übergibt die Steuerung des Programmablaufs beim rekursiven Aufruf an die gerufene Prozedur. Nach Erledigung ihrer Aufgabe gibt diese – zusammen mit ihren Ergebnissen – die Ablaufsteuerung an die rufende Prozedur zurück, die sodann mit der Ausführung der hinter dem Aufruf kommenden Anweisungen fortfährt.“[30] Wie dies im konkreten Fall aussieht, werden wir im nächsten Abschnitt sehen.

1.3.2 Programmiertechnische Umsetzung[31]

Wie Rekursion programmtechnisch funktioniert, lässt sich sehr schön an folgendem Pascal-Programm demonstrieren, das die Rekursion in ihrer einfachsten Form darstellt. Mit Hilfe der Standardfunktion *readkey* wird ein Zeichen von der Tastatur eingelesen, ohne dass es auf dem Bildschirm dargestellt wird. Dieser Vorgang wird so lange wiederholt, bis das Zeichen ’_’ eingegeben wird:

```
program Rekursion;
uses crt;   {zum Empfangen von Bildschirmbefehlen}

  procedure liesZeichen;
  var z: Char;
  begin
    z:= readkey;
    if  z <>  '_'  then liesZeichen;
    write (z);
  end;

begin
  clrscr;  {Bildschirm löschen}
  liesZeichen;
  repeat until keypressed; {warte auf Tastendruck / Anschauen der Ausgabe}
end.
```

[29] Vgl. Baumann (1993) 14.

[30] Baumann (1993) 22.

[31] Vgl. Wehrheim 7ff.

Das Programm gibt anschließend die eingegebene Zeichenfolge in der umgekehrten Reihenfolge aus. Gibt man beispielsweise nacheinander die Zeichen ’A’, ’B’, ’C’ und ’_’ ein, so wird die Zeichenkette ’_CBA’ auf dem Bildschirm ausgegeben.

Wie ist dies zu erklären?

Bei jedem eingegebenen Zeichen, das ungleich ’_’ ist, ruft sich die Prozedur *liesZeichen* selbst auf, noch bevor die *write*-Anweisung ausgeführt wird. Wird allerdings das Zeichen ’_’ eingegeben, erfolgt kein weiterer Aufruf; die Rekursion bricht ab. Nun wird die *write*-Anweisung zum ersten Mal ausgeführt, und das Zeichen ’_’, das gerade in der Variablen *z* gespeichert ist, wird ausgegeben. Damit ist der letzte Prozeduraufruf vollständig abgearbeitet. Anschließend kann der vorletzte Prozeduraufruf fortgesetzt werden, und es erfolgt die Ausgabe des Zeichens ’C’. Damit ist auch dieser Prozeduraufruf vollständig abgearbeitet, und es wird nun der Prozeduraufruf davor fortgesetzt usw. Erst wenn auf diesem Wege alle Prozeduraufrufe vollständig abgearbeitet sind, ist das Programm beendet.

Die folgende Abbildung verdeutlicht diese Verfahrensweise:

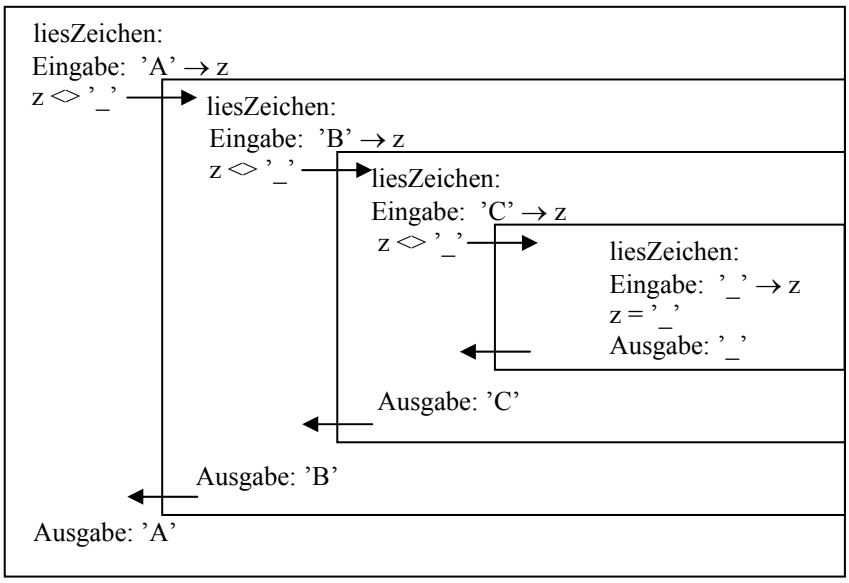


Abb. 6: Verfahrensweise der Prozedur liesZeichen

Während des "untersten" rekursiven Aufrufs von *liesZeichen* sind drei offene rekursive Aufrufe gestapelt und werden bei der Rückkehr aus der Rekursion wieder abgebaut.

Es fällt hierbei auf, dass die einzelnen Zeichen in einer einzigen Variable gespeichert werden. Dies ist dadurch zu erklären, dass die Variable *z* eine lokale Variable bezüglich der Prozedur *liesZeichen* ist. Für lokale Variablen werden erst beim Prozeduraufruf selbst Speicherzellen zur Verfügung gestellt, die erst dann wieder frei gegeben werden,

wenn die Prozedur vollständig abgearbeitet ist. So werden also bei jedem Prozeduraufruf für die Variable *z* neue Speicherzellen verwendet. Das vorgestellte Programm belegt demnach in jeder Rekursionsstufe andere Speicherzellen, die allerdings auf jeder Stufe den gleichen Namen haben.

Bei der rekursiven Programmierung spielen lokale Variable eine wichtige Rolle. Für die lokalen Variablen einer rekursiven Prozedur werden bei jedem Aufruf neue Speicherzellen zur Verfügung gestellt, die auf jeder Rekursionsstufe den gleichen Namen haben.[32]

Auch Werteparameter von Prozeduren und Funktionen werden bei der rekursiven Programmierung wie lokale Variablen verwaltet.[32]

[32] Siehe Wehrheim 9.

2 REKURSION IM INFORMATIKUNTERRICHT

2.1 Didaktische Überlegungen: Rekursion im unterrichtlichen Zusammenhang

Folgende Mindmap gibt das Auftreten des Themas „Rekursion“ im Lehrplan der gymnasialen Oberstufe wieder.

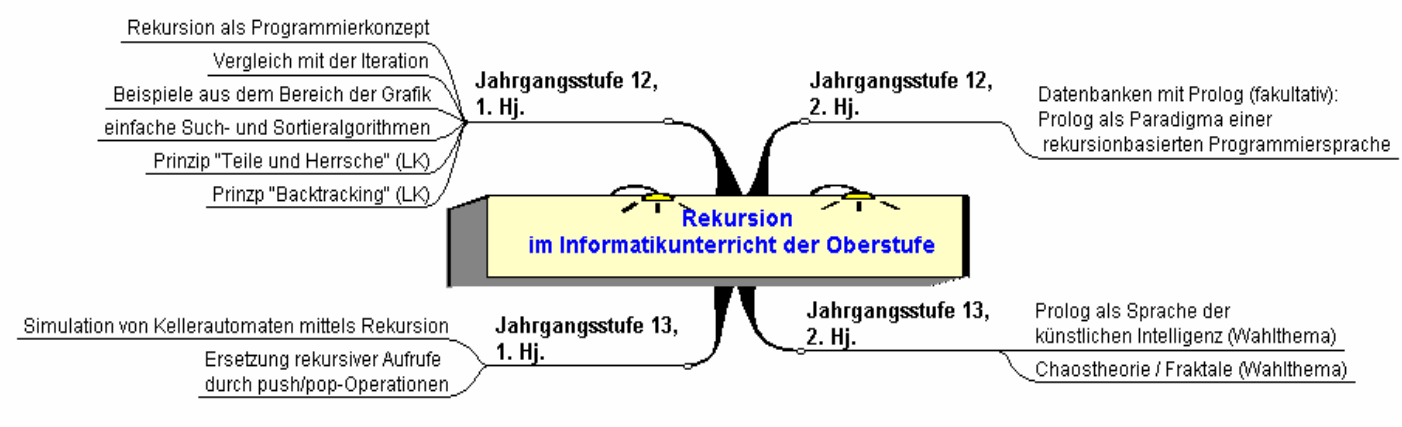


Abb. 7: Das Thema „Rekursion“ im Lehrplan der gymnasialen Oberstufe

Im Sinne des Spiralcurriculums wird das Thema Rekursion in den vier Halbjahren 12/I bis 13/II aufgegriffen; zum Teil als fakultativer Unterrichtsinhalt. Es stellt, zumindest im Leistungskurs, eine Leitlinie des Informatikunterrichts der Jahrgangsstufen 12 und 13 dar und „wird spiralartig mehrfach durchlaufen, das heißt in den Kursinhalten aller Jahrgangsstufen treten die Ziele und Inhalte mit zunehmender Komplexität und höherem Abstraktionsniveau wieder auf.“[33]

Im weiteren Verlauf dieser Arbeit liegt das Hauptaugenmerk auf den Inhalten der Jahrgangsstufe 12/I, wo der Grundstein zum Verständnis der Rekursion gelegt wird.

[33] Hess. Kultusministerium (Lehrplan Informatik) 3.

2.2 Methodische Überlegungen: Einstieg in das Thema „Rekursion“

Die rekursive Denkweise sollte den Schülerinnen und Schülern zunächst programmiersprachenunabhängig nahe gebracht werden. Dies kann durch die Betrachtung einiger der in Kapitel 1 genannten Beispiele geschehen. Dabei muss jedoch darauf eingegangen werden, dass viele dieser Beispiele das eigentliche Wesen der Rekursion in der Informatik nicht verdeutlichen. Man sollte daher unbedingt solche Beispiele mit einbeziehen, die den Problemlösecharakter, die sukzessive Vereinfachung und schließlich den Rekursionsausstieg veranschaulichen.

Bezüglich der programmtechnischen Umsetzung sollte als Einstiegsbeispiel in das Thema „Rekursion“ ein Problem gewählt werden, dessen Struktur eine rekursive Lösung herausfordert, die einerseits nicht trivial, aber andererseits nicht zu umfangreich bzw. nicht zu schwierig ist. Ferner sollte die rekursive Lösung (relativ) effizient sein, da sonst die Notwendigkeit der neuen Programmstruktur den Schülerinnen und Schülern nicht einsichtig gemacht werden kann.[34]

In vielen Informatik-Lehrwerken werden rekursiv definierte mathematische Funktionen als Einstiegsbeispiel bemüht, am häufigsten die Fakultätsfunktion (vgl. Abschnitt 1.2.3). Diese ist – abgesehen von der Eleganz der rekursiven Definition - jedoch kein geeignetes Einstiegsbeispiel, da sich die Fakultät auch leicht iterativ berechnen lässt und so die Notwendigkeit der Rekursion nicht gegeben ist. Gegen den Einstieg mittels einer mathematischen Funktion spricht auch, dass die Gefahr besteht, dass die Diskussion des mathematischen Problems zu sehr in den Vordergrund gerät.

Ebenfalls ungeeignet in diesem Zusammenhang erscheint ein Backtracking-Algorithmus (z.B. Labyrinth-Probleme, Weg des Springers, Acht-Damen-Problem, siehe Kapitel 4), weil hier neben dem Verfahren der Rekursion gleichzeitig das des „trial and error“ (siehe Abschnitt 4.4) eingeführt werden müsste.

---

[34] Vgl. Knöß 83.

2.2.1 Beispiele aus der Grafik als Einführung in das Thema Rekursion

BAUMANN (1985) unterstützt die Idee, mit Hilfe einer rekursiven Turtle-Grafik[35] in die Programmierung rekursiver Algorithmen einzuführen. Um die Schülerinnen und Schüler „Idee und Erscheinungsformen der Rekursion erfahren zu lassen, eignen sich [...] besonders gut Beispiele aus der Grafik, insbesondere aus dem Gebiet der so sogenannten *Monsterkurven* (das sind Kurven mit unendlicher Länge und endlichem Inhalt). Die Schüler sind angesichts des einfachen Bildungsgesetzes über die Komplexität der Kurven verblüfft. Mehr noch: nirgendwo sonst wird das Wesen der Rekursion, nämlich die *Selbstähnlichkeit* so unmittelbar augenscheinlich; jeder – auch der kleinste Ausschnitt hat bereits alle Merkmale der ganzen Kurve“[36].

Hierbei werden elementare Kenntnisse der grundlegenden Kontrollstrukturen und einfacher Grafik-Prozeduren vorausgesetzt, sowie der Begriff der Rekursion, wie er im Unterricht an den üblichen Trivialbeispielen (Fakultät, Summation, Fibonacci-Folge etc.) eingeführt wird.“[37]

Auch der Lehrplan sieht eine Behandlung von Grafik-Beispielen vor mit dem Hinweis, dass sie „eine unmittelbare Rückmeldung über die Korrektheit der Lösung“ ermöglichen.[38] Die grafische Rekursion gibt den Schülerinnen und Schülern außerdem einen außermathematischen Einblick in Anwendungsgebiete der Informatik, welche hier im weitesten Sinne der Computerkunst zuzuordnen sind.

Im Folgenden möchte ich als Beispiel einer Turtle-Grafik die sogenannte Koch-Kurve und deren Implementierung vorstellen. Auf der Grundlage der drei Standardbefehle zur Erzeugung einer Turtle-Grafik *Gehe n Zentimeter vorwärts (VW)*, *Drehe um d° nach links (DL)* und *Drehe um d° nach rechts (DR)* ist eine schrittweise Entwicklung einer rekursiv angelegten Befehlsfolge möglich, welche zu einer beliebig genauen

[35] Diese Bezeichnung kommt von einer Schildkröte, die sich entweder vorwärts bewegt oder eine Drehung macht und dabei eine Spur hinterlässt. [Gumm 624]

[36] Baumann 1985, 30.

[37] Ebenda.

[38] Hess. Kultusministerium (Lehrplan Informatik) 13.

Annäherung an die Koch-Kurve, die auch Schneeflockenkurve oder Eisblume genannt wird, führt.

Die Konstruktionsregel ist, dass jedes Geradenstück durch vier neue Geradenstücke mit je einem Drittel der Originallänge in folgender Anordnung ersetzt wird:

Diese Ersetzung wird mehrfach rekursiv wiederholt und führt damit zu folgendem Ergebnis:

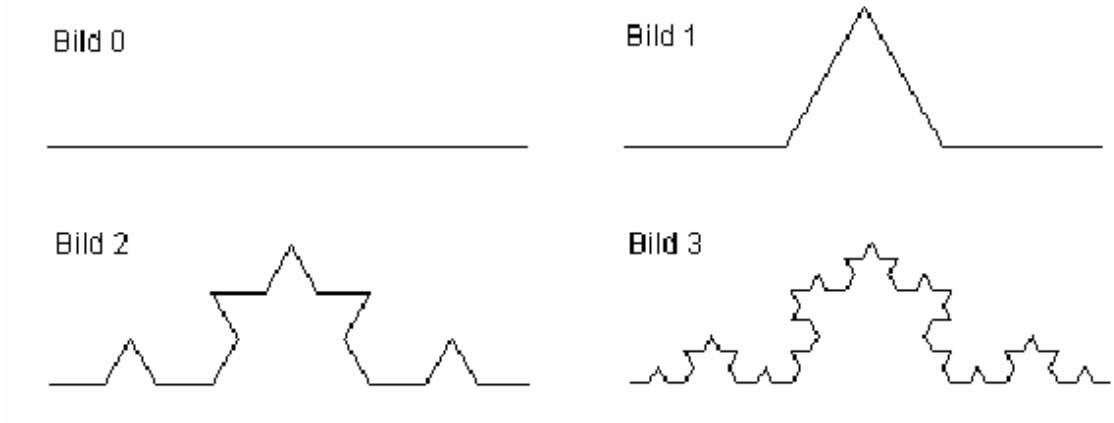


Abb. 8: Konstruktion der Koch-Kurve[39]

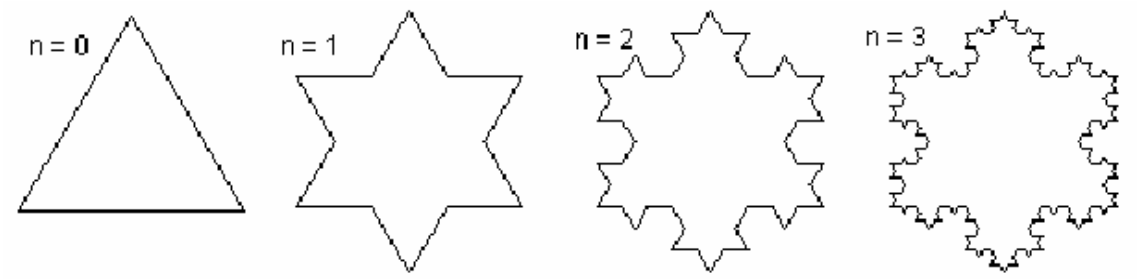
Die Prozedur, mit der man das Bild, das durch *n* Ersetzungsschritte charakterisiert ist, erzeugen kann, lautet in Delphi:

```
procedure TForm1.Kochkurve(Nummer: integer; Strichlaenge: real);
begin
  if Nummer=1 then Turtle.VW(Strichlaenge)
              else begin
                     Kochkurve(Nummer-1, Strichlaenge/3);
                     Turtle.DL(60);
                     Kochkurve(Nummer-1, Strichlaenge/3);
                     Turtle.DR(120);
                     Kochkurve(Nummer-1, Strichlaenge/3);
                     Turtle.DL(60);
                     Kochkurve(Nummer-1, Strichlaenge/3);
                   end;
end;
```

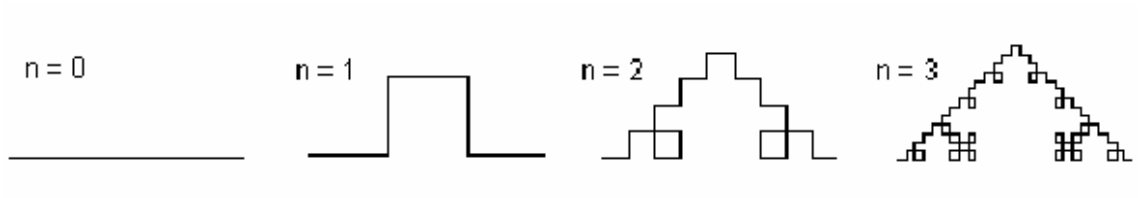
[39] Abb. aus: Röhner (Rekursive Grafiken) 1.

Der Prozess der Streckenersetzung wurde hier mit der Strecke aus Bild 0 initiiert. Man kann aber auch ganze Streckenzüge als Initiator benutzen:

Koch-Kurve mit gleichseitigem Dreieck als Initator:



Kreuzstich mit Strecke als Initiator:



Kreuzstich mit Quadrat als Initiator:

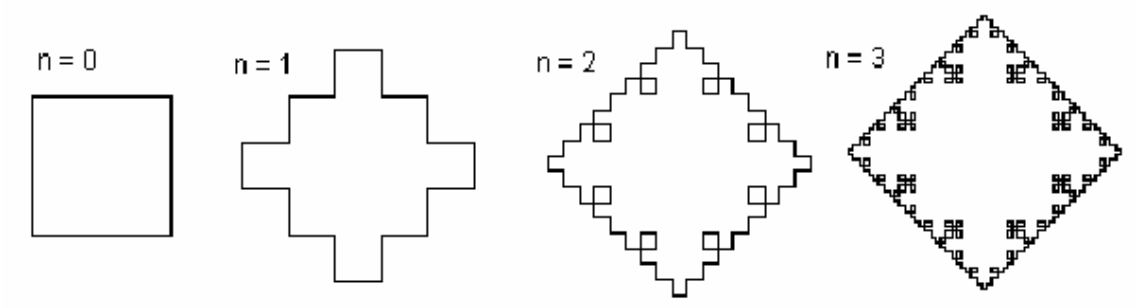


Abb. 9: Streckenzüge als Initiatoren

Zum Beispiel erhält man den Kreuzstich mit Quadrat als Generator durch eine Zeichenprozedur für ein Quadrat, bei der man die Draw-Befehle durch den Aufruf der rekursiven Generatorprozedur ersetzt.[40]

[40] Abbildungen und Erläuterungen bei Röhner (Rekursive Grafiken) 4.

Schließlich können die Schülerinnen und Schüler die neuen Kenntnisse dazu nutzen, eigene rekursive Grafiken und deren zugehörige Algorithmen zu entwickeln.

So faszinierend diese Grafiken auch sind, so darf man jedoch nicht außer Acht lassen, dass sie den *Problenlösecharakter* der Rekursion nicht wiedergeben! Die Programme verfolgen – abgesehen von der Ästhetik der erzeugten Grafiken – keinen weiteren Zweck. Unter diesem Gesichtspunkt eignet sich das Beispiel der Turtle-Grafik trotz ihrer Anschaulichkeit nur bedingt zur Einführung in die Rekursion.

2.2.2 Die „Türme von Hanoi“ als geeignetes Einstiegsbeispiel

Ein Beispiel, das die oben genannten Anforderungen an ein Einstiegsbeispiel erfüllt, sind die „Türme von Hanoi“. Die Gründe dafür werden weiter hinten genannt. Doch zunächst möchte ich das Problem vorstellen:

Die Türme von Hanoi

Bei den Türmen von Hanoi sind drei senkrechte Stangen auf einem Holzbrett befestigt. Auf einer der Stangen steckt am Anfang eine Anzahl unterschiedlich großer, in der Mitte durchbohrter Scheiben. Sie sind nach abnehmender Größe von unten nach oben übereinander gestapelt. Das Ziel des Spiels ist, den Turm soll von Stange A nach Stange B umzustapeln, wobei Stange C als Zwischenablage benutzt werden darf.

Die Spielregeln sind sehr einfach:

1. Es darf immer nur eine Scheibe bewegt werden.
2. Man darf nie eine größere Scheibe auf eine kleinere legen.

Die Schülerinnen und Schüler können diese Aufgabe leicht nachstellen, indem sie z. B. Spielkarten unterschiedlicher Werte oder Münzen unterschiedlicher Größe als Scheiben verwenden und die Stäbe durch Notizzettel darstellen. Der Anfang des Spiels ist schon vorgegeben: Da nur die kleinste Scheibe zugänglich ist, kann nur sie versetzt werden (siehe Abb. 10). Als nächstes ist nur das Versetzen der zweitkleinsten Scheibe sinnvoll; diese muss auf die leere Stange gesetzt werden, da man sie nicht auf die kleinste Scheibe legen darf. Da ein erneutes Bewegen der zweiten Scheibe im nächsten Zug nur den vorherigen Zustand wieder herstellen würde, muss dieser Zug mit der ersten Scheibe ausgeführt werden. Nun stehen die Schülerinnen und Schüler vor der Entscheidung, auf welche Stange sie die kleinste Scheibe stecken sollen.

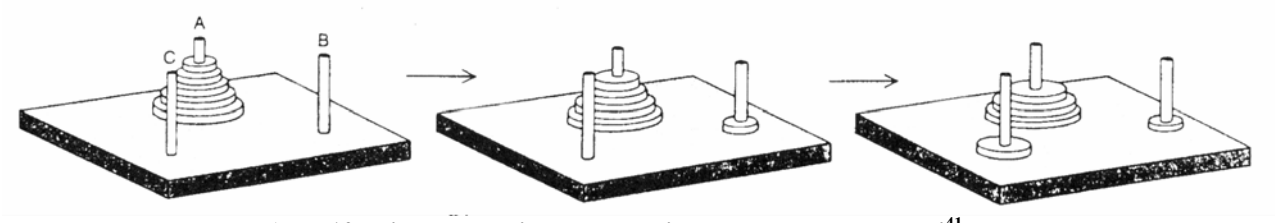


Abb. 10: Die ersten beiden Züge bei den Türmen von Hanoi[41]

Von hier an muss eine lange Folge von Zügen realisiert werden, um zur Lösung zu gelangen. Selbst wenn man keinen Fehler macht, braucht man, wie wir noch sehen werden, 2^n-1 Züge, um einen kompletten Ausgangsturm mit n Scheiben auf eine andere Stange zu versetzen. „Diese erstaunlich lange Überführungszeit selbst bei einem Turm mit verhältnismäßig wenigen Scheiben wird sehr schön durch die folgende kleine Legende illustriert (zitiert nach dem klassischen Rätselbuch ‚Mathematical Recreations and Essays' von W. W. Rouse Ball):

> ‚In dem großen Tempel von Benares ... unter der Kuppel am Mittelpunkt der Welt ruht eine Messingplatte, in die drei diamantene Nadeln eingelassen sind. Jede ist eine Elle hoch und so schmal wie der Körper einer Biene. Auf eine dieser Nadeln steckte Gott am Tage der Schöpfung vierundsechzig Scheiben aus purem Gold, die größte ganz unten auf die Messingplatte und die anderen mit abnehmender Größe bis zur kleinsten ganz oben. Das ist der Turm des Brahma. Tag und Nacht versetzen die Mönche seither unermüdlich Scheibe um Scheibe von Nadel zu Nadel, gemäß dem festen und unabänderlichen Gesetz des Brahma. Dieses gebietet, daß ein jeder Mönch jeweils immer nur eine Scheibe versetzen darf und sie so auf eine Nadel stecken muß, daß keine kleinere Scheibe unter ihr liegt. Wenn dann alle vierundsechzig Scheiben von der Nadel, auf die Gott sie am Schöpfungstag gelegt hat, auf eine andere Nadel überführt sind, werden Turm, Tempel und Brahmanen gleichermaßen zu Staub zerfallen und die Welt mit einem Donnerschlag verschwinden.'‹‹[42]

Dass die Welt noch steht, beweist, dass die Lösung der Aufgabe doch recht lange dauert: Selbst wenn die Mönche in jeder Sekunde eine Scheibe umstecken könnten, brauchten sie mehr als 500 Milliarden Jahre, um den ganzen Turm auf einer anderen Nadel neu zu errichten.

Wenn die Schülerinnen und Schüler einige Zeit versucht haben, das Problem für einen Turm aus fünf Scheiben zu lösen, werden sie sicherlich bemerken, dass immer wieder kleinere Türme auftauchen: Türme aus zwei Scheiben bilden sich recht oft, manchmal auch solche aus drei oder gar vier Scheiben. Auch wenn sie noch keine klare Vorstellung von der Lösung haben, bemerken sie vielleicht gewisse Regelmäßigkeiten bei der Bildung solcher kleineren Türme.

---

[41] Abb. aus Dewdney 8.

[42] Dewdney 8.

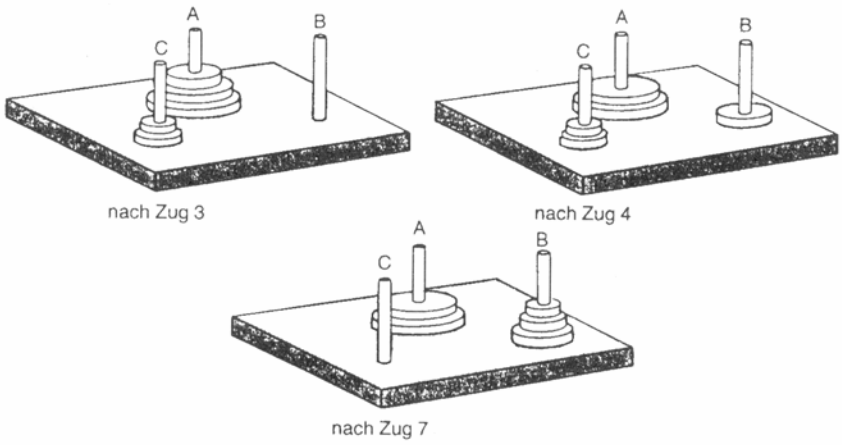


Abb. 11: Ausschnitt aus der rekursiven Lösung für die Türme von Hanoi[43]

Sie erkennen bestimmt auch, dass die unterste Scheibe nur versetzt werden kann, wenn diese zum einen frei auf einem Platz liegt und zum anderen die Zielstange als neuer Platz für sie frei ist. Dies ist aber nur möglich, wenn alle anderen *n*-1 Scheiben auf dem Hilfsstab stecken, und zwar gemäß der 2. Regel der Größe nach geordnet.

So bekommen die Schülerinnen und Schüler wahrscheinlich schon eine Ahnung, wie die Lösung des Problems aussieht: Letztlich wird der Stapel der oberen *n*-1 Scheiben von der Ausgangsstange zur Hilfsstange überführt, dann die *n*-te, also größte, Scheibe von der Ausgangsstange zur Zielstange und schließlich wieder der Stapel aus *n*-1 Scheiben von der Hilfsstange zur Zielstange. Und nun liegt der Gedanke an die Rekursion nahe: Das Problem des Umsetzens von *n* Scheiben ist auf das einfachere Problem des Umsetzens von *n*-1 Scheiben zurückgeführt worden.

Bei der **Lösung** schließlich wird diese Vereinfachung rekursiv angewendet:

- Wenn man *n* Scheiben umstapeln soll, nimmt man an, dass man jemanden kennt, der das Problem für *n*-1 Scheiben lösen kann.
- Diese Person veranlasst man nun dazu, *n*-1 Scheiben umzuordnen, und zwar auf den Hilfsstab. Danach legt man selbst die übrig bleibende Scheibe auf den Zielstab und veranlasst die Person, die schon einmal *n*-1 Scheiben umgeordnet hat, dies nochmals zu tun, diesmal jedoch auf die große Scheibe auf den Zielstab.

[43] Abb. aus Dewdney 9.

- Von der Person, die *n*-1 Scheiben umordnen kann, nimmt man an, dass sie wiederum eine Person kennt, die *n*-2 Scheiben umsortieren kann. So muss die erste Person nur eine Scheibe an eine andere Position legen, den Rest erledigt die zweite Person.
- Von dieser und allen weiteren Personen nimmt man auch an, dass sie eine Person kennen, die das Problem für jeweils eine Scheibe weniger lösen kann (Rekursion!). Die letzte Person in dieser Reihe hat nur noch eine Scheibe umzulegen (Rekursionsabbruch), und das geht ja nun wirklich einfach.[44]

Dieses Prinzip des Delegierens als wichtiges und grundlegendes Schema einer rekursiven Problemlösung ist uns schon eingangs beim Beispiel "Kohlen holen" begegnet: Ebenso wie ein Junge maximal einen Eimer Kohle schleppen muss, so muss bei den Türmen von Hanoi ein Mensch nur maximal eine Scheibe bewegen. Die ganze restliche Arbeit wird delegiert an Menschen, die wieder delegieren. Dabei wird allerdings vorausgesetzt, dass diese auch bis zur endgültigen Lösung des Problems gefunden werden können.

Das Lösungsverfahren zeigt, warum die Lösung nahezu 2^n Schritte erfordert: Jedesmal, wenn ein Turm versetzt werden soll, muss der um eine Scheibe kleinere Turm zweimal versetzt werden.

Der **Quelltext** für die rekursive Prozedur ist noch kürzer als die Erklärung:[45]

```
procedure Bewege_Scheiben(Anzahl:Integer;links,mitte,rechts:Char);
begin
  if Anzahl = 1
    then
      begin
        Form1.ListBox1.Items.Add(links+'->'+mitte);
        Inc(BewegungsAnzahl);
      end
    else
      begin
        Bewege_Scheiben(Anzahl-1,links,rechts,mitte);
        Bewege_Scheiben(1,links,mitte,rechts);
        Bewege_Scheiben(Anzahl-1,rechts,mitte,links);
      end;
end;
```

Das Programm enthält seinen rekursiven Aufruf sogar doppelt, einmal zum Transportieren eines Stapels auf den Hilfsstab zur Zwischenlagerung und einmal zum

[44] Bähnisch 178.

[45] Ebenda.

Transportieren desselben Stapels auf den Zielstab. Dabei werden bei der rekursiven Ausführung die Rollen von Ziel- und Hilfsstab ständig vertauscht.

Ein Lösungsweg für Menschen[46]

Dieses Lösungsverfahren, das sich wie ein Baum gnadenlos immer weiter in Äste und Unteräste verzweigt, ist für einen Menschen in der Praxis allerdings schwer zu überblicken, da man sich immer genau merken können muss, an welcher Stelle der Lösungsabarbeitung man gerade ist. Es gibt allerdings für die Türme von Hanoi auch einen Lösungsweg[47], der ganz einfach auszuführen ist: Man kommt zur Lösung, indem man lediglich abwechselnd die beiden folgenden Züge ausführt:

1. Stecke die kleinste Scheibe auf die im Uhrzeigersinn nächste Stange.
2. Versetze irgendeine Scheibe außer der kleinsten, ohne die Regeln zu verletzen..

Der zweite Schritt sieht zwar beliebig aus, ist aber durchaus eindeutig, denn es existiert dafür immer nur genau ein erlaubter Zug.

Diese Lösung zeigt, dass man bei den Türmen von Hanoi auch ohne Rekursion auskommt. Ein iteratives Programm, das immer dieselben Schritte innerhalb einer einfachen Schleife wiederholt, führt auch zur Lösung. Allerdings ist eine iterative Implementation nicht so einfach zu finden und auch nicht so elegant wie die rekursive.

[46] Dewdney 10.

[47] von Buneman (Universität von Pennsylvania) und Levy (AT&T Bell Laboratories)

Eignung der „Türme von Hanoi“ als Einstiegbeispiel

Das Beispiel „Türme von Hanoi“ ist sehr gut als Einstiegsbeispiel für die Rekursion geeignet, und zwar aus folgenden Gründen:[48]

- Es wird eine Problemlösung eingefordert.
- Es zielt direkt auf die Programmierung.
- Da es praktisch ausgeschlossen ist, dass die Schülerinnen und Schüler eine iterative Lösung finden, wird eine rekursive Lösung geradezu herausgefordert.
- Der Lösungsalgorithmus ist verblüffend kurz. Dadurch wird das Nachdenken über dessen Wirkung motiviert und erleichtert.
- Da außer den Parametern keine zusätzlichen, nur im Prozedurrumpf deklarierten Objekte vorkommen, wird das Nachvollziehen der Abarbeitung erleichtert, denn durch diese Reduktion können sich die Schülerinnen und Schüler auf die Reihenfolge der rekursiven Aufrufe konzentrieren. Weiterhin wird bei der Besprechung weiterer Beispiele mit zusätzlichen lokalen Objekten eine Vertiefung nach dem Spiralprinzip ermöglicht.
- Der Spiel-Charakter der Aufgabe motiviert die Schülerinnen und Schüler.
- Die Aufgabe ist handlungsorientiert: Man kann das Problem mit einfachen Mitteln nachspielen bzw. mit Java-Applets verdeutlichen.

Das Problem „Türme von Hanoi“ ist ein Beispiel, das nicht nur Stoff für tiefgründige Betrachtungen bietet, es macht auch Spaß und versetzt den Neuling „in jenen angenehmen Zustand der Verwirrung, der das Zeichen des Eintritts in die Sphäre des abstrakten Denkens ist.“[49]

[48] Vgl. Knöß 84.

[49] Dewdney 8.

2.3 Programmiersprachen, die sich der Rekursion bedienen (müssen)

Im Rahmen theoretischer Betrachtungen über Programmierung und Programmiersprachen ist der folgende, in der Informatik bewiesene, Satz von grundlegender Bedeutung:

Jeder iterative Algorithmus lässt sich in einen rekursiven Algorithmus umwandeln und umgekehrt.

Diese Erkenntnis ermöglicht die Existenz von Programmiersprachen, die in ihrem Grundkonzept nur über das Hilfsmittel Rekursion, nicht aber über Iterationen in Form von Wiederholungsanweisungen verfügen[50].

Als Beispiele für Programmiersprachen, die die Rekursion als grundlegendes Konzept verwenden, möchte ich hier die deklarativen Programmiersprachen LISP (funktionale Sprache) und PROLOG (logische Sprache) kurz vorstellen. In diesen Sprachen gibt es keine Variablen, die einen Zustand speichern können; daher werden wiederholte Vorgänge mittels Rekursion, nicht mittels Schleifen programmiert.

Ein in einer deklarativen Sprache geschriebenes Programm „beschreibt die allgemeinen Eigenschaften von Objekten und ihre Beziehungen untereinander. Das Programm beschreibt zunächst nur das Wissen zur Lösung eines Problems.“[51]

2.3.1 Die funktionale Programmiersprache LISP

Ein Programm in einer funktionalen Programmiersprache wie LISP „wird als eindeutige Abbildung (Funktion) aus der Menge der Eingabedaten auf die Menge der Ausgabedaten betrachtet und stellt sich als Zusammensetzung aufeinander bezogener, einfacher Funktionen dar. Es besteht die Möglichkeit, dass sich das Programm selbst aufruft.“[52]

---

[50] Siehe Damann 223.

[51] Engelmann 38.

[52] Engelmann 38.

Wollte man in LISP die Rekursion vermeiden, müssten alle benötigten Variablen ständig als Parameter übergeben werden (siehe das folgende Lisp-Beispiel: Iterative Fakultätsberechnung). Die Ineffizienz von rekursiven Varianten spielt in funktionalen Programmiersprachen oft keine Rolle, weil hier die Flexibilität mehr gefragt ist. Lisp, Logo, ML, Miranda oder Haskell sind Beispiele für funktionale Programmiersprachen.

Rekursive Fakultätsberechnung (in LISP)

```
(define (fakultät n)
    (if (= n 1)
        1
        (* n (fakultät (- n 1)))))
```

Funktionsaufruf des linear rekursiven Prozesses zur Berechnung von 6!

```
(fakultät 6)
(* 6 (fakultät 5))
(* 6 (* 5 (fakultät 4)))
(* 6 (* 5 (* 4 (fakultät 3))))
(* 6 (* 5 (* 4 (* 3 (fakultät 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (fakultät 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Iterative Fakultätsberechnung (in LISP)

```
(define (fakultät n)
    (fakultät-iterativ 1 1 n))
(define (fakultät-iterativ produkt zähler max-zähler)
  (if (> zähler max-zähler)
        produkt
        (fakultät-iterativ (* zähler produkt) (+ zähler 1) max-
zähler)))
```

Funktionsaufruf des linear iterativen Prozesses zur Berechnung von 6!

```
(fakultät 6)
(fakultät-iterativ   1 1 6)
(fakultät-iterativ   1 2 6)
(fakultät-iterativ   2 3 6)
(fakultät-iterativ   6 4 6)
(fakultät-iterativ  24 5 6)
(fakultät-iterativ 120 6 6)
(fakultät-iterativ 720 7 6)
720 [53]
```

[53] Programme aus. http://www.fim.uni-linz.ac.at/lva/rus/Rekursion/Rekursion.htm.

2.3.2 Die logische Programmiersprache PROLOG

Wie LISP kennt auch PROLOG keine expliziten Kontrollstrukturen für die Wiederholung und keine Wertzuweisung an Variablen. Lediglich Sequenz, Rekursion und Prozeduraufruf stehen zur Verfügung. Im Gegensatz zu LISP ist in PROLOG eine Fallunterscheidung nicht möglich.[54]

Ein Programm, das in der logischen Programmiersprache PROLOG (abgeleitet von **pro**gramming in **log**ic) geschrieben ist, ist die Niederschrift von Fakten und Regeln, womit der Computer neue Fakten gewinnt und das Problem gelöst wird.[55]

Das folgende Beispiel soll dies verdeutlichen:

Es soll ein Prädikat zur Bestimmung der Vorfahren in einer Familienbeziehung angegeben werden. Dazu wird vorfahr(X,Y) wie folgt definiert: X ist ein Vorfahr von Y, falls X ein Elternteil von Y ist. Damit findet man aber nur die direkten Vorfahren. Nun wird die Rekursion zur Ermittlung weiterer Vorfahren genutzt: X ist ebenfalls ein Vorfahr von Y, falls X ein Elternteil von Z ist und Z ein Vorfahr von Y ist.

```
vorfahr(X,Y):-elternteil(X,Y).
vorfahr(X,Y):- elternteil(X,Z), vorfahr(Z,Y).
```

Die erste *vorfahr*-Klausel ist nicht rekursiv und sorgt für die Terminierung der Rekursion. Die zweite *vorfahr*-Klausel enthält den rekursiven Aufruf. [56]

[54] Siehe Röhner (Informatik mit Prolog) 3.

[55] Engelmann 38.

[56] Röhner (Informatik mit Prolog) 9.

Die Fakultätsberechnung in PROLOG

Zum Vergleich zu LISP wird hier die rekursive Fakultätsberechnung in PROLOG wiedergegeben. Die erste Klausel stellt den definitorischen Sonderfall und die Abbruchbedingung dar. Dies ist der Fall, wenn die Fakultät von 0 errechnet werden soll, die per Definition 1 ist.

```
fak(0,1).
fak(X, Fakultaet):-
      X > 0,
      Y is X - 1,
      fak(Y, Z),
      Fakultaet is Z * X.
```

Z wird erst im allerletzten Aufruf von fak(...) initialisiert, nämlich wenn Y = 0 ist, und zwar mit 1 (durch die erste Klausel).

Doppelte Rekursion in PROLOG

Ein Beispiel der *doppelten Rekursion* zeigt das folgende PROLOG-Programm, welches die Summe der ganzzahligen Knotenwerte eines binären Baums berechnet.

```
baumsumme(nil, 0).
baumsumme(baum(Linker, Knoten, Rechter), S) :-
     baumsumme(Linker, S1),
     baumsumme(Rechter, S2),
     S is S1 + S2 + Knoten.
```

Hier wurde die Struktur des listenförmig aufgebauten Baums verwendet.

PROLOG ist im Lehrplan der gymnasialen Oberstufe als fakultativer Unterrichtsinhalt aufgeführt: als Paradigma einer rekursionsbasierten Programmiersprache im Rahmen der Behandlung von Datenbanken (Jahrgangsstufe 12/II) und als Sprache der künstlichen Intelligenz (Wahlthema in in 13/II).

3 REKURSION VERSUS ITERATION

Iterativ arbeiten ist menschlich, rekursiv arbeiten ist göttlich.

Anonym

Rekursion ist Wiederholung durch Verschachtelung, Iteration Wiederholung durch Aneinanderreihung, genauer: Berechnung durch wiederholtes Anwenden einer Operation in einer Schleife. „Rekursion und Iteration bilden eines der vielen Gegensatzpaare in der Informatik, eine Art Yin und Yang der Schleifenprogrammierung.“[57]

3.1 Probleme, die sich durch Rekursion ergeben

3.1.1 Hoher Speicherplatzbedarf

In einem rekursiv implementierten Programm benötigt jede Prozedur P, die einen erneuten Rekursionsaufruf startet, nicht nur Speicherplatz für ihre lokalen Variablen, sondern zusätzlich auch für die Speicherung des aktuellen Stands der Berechnung, damit dieser nach Beendigung des erneuten Aufrufs von P für die Fortsetzung der vorangegangenen Aktivierung wieder zur Verfügung steht.[58]

Wie aus den obigen Beispielen der rekursiven und iterativen Fakultäts- bzw. Fibonaccizahl-Berechnung ersichtlich ist, wächst die zu speichernde Information mit dem Prozessfortschritt in der rekursiven Variante. Bei der iterativen Variante bleibt sie konstant, da hier bereits die einzelnen Variablen den Zustand des Berechnungsprozesses wiederspiegeln.

Der Speicherbereich, der sogenannte *stack* (=Stapel), kann bei großer Rekursionstiefe schnell anwachsen und den verfügbaren Speicher ausfüllen, was zu einem Programmabsturz führen kann (Stackoverflow). Daher ist es wichtig, dass bei praktischen Anwendungen die größte Tiefe der Rekursion nicht nur endlich, sondern sogar klein ist.

---

[57] Dewdney 11.

[58] Wirth 151.

3.1.2 Unendliche Rekursion

Wie Wiederholungsanweisungen bergen auch rekursive Prozeduren die Gefahr nicht abbrechender Ausführung. Wichtig ist, dass der rekursive Aufruf einer Prozedur von einer Bedingung abhängt, die irgendwann nicht mehr erfüllt ist. Ansonsten kommt es zu einer unendlichen Rekursion. Um dies zu umgehen, könnte man bei jedem Funktionsaufruf die Rekursionstiefe als Parameter übergeben. So wäre man mit Hilfe dieses Sicherheitszählers in der Lage, die Rekursion bei einer bestimmten Tiefe zu unterbrechen.

3.2 Grundprinzipien und Eigenschaften im Vergleich

Die rekursive Programmierung eignet sich besonders, wenn das zu lösende Problem oder die auszuführende Funktion rekursiv dargestellt ist. Allerdings ist es aus den eben genannten Gründen nicht immer der beste Weg, einen rekursiven Algorithmus auch in einem rekursiven Programm umzusetzen. „Tatsächlich hat die Erklärung des Konzepts rekursiver Algorithmen anhand von ungeeigneten Beispielen wesentlich zur Entstehung einer weitverbreiteten Abneigung und Antipathie gegen die Rekursion in der Programmierung beigetragen und zur Gleichsetzung der Rekursion mit Ineffizienz geführt.“[59]

Als einfaches Beispiel möchte ich hier die Fakultätsfunktion nennen (vgl. Abschnitt 1.2.3):[60]

$$f(0) = 0! = 1;$$

$$f(n) = n! = f(n\text{-}1) \cdot n$$

Die rekursive Implementierung in Delphi lehnt sich an die Definition an:

```
function Faku(n: integer): LongInt;
begin
  if n=0 then Faku:=1
         else Faku:=Faku(n-1)*n;
end;
```

[59] Wirth 152.

[60] Vgl. Wirth 152.

Allerdings kann hier die Rekursion durch eine einfache Wiederholung, z. B. in Form von einer for-Schleife, ersetzt werden:

```
function Faku(n: integer): LongInt;
var i, Faktor: LongInt;
begin
    Faktor := 1;
    for i:=1 to n do
      Faktor:=Faktor*i;
    Faku := Faktor;
end;
```

Wie man sieht, lässt sich leicht eine iterative Lösung angeben, obwohl die mathematische Definition von $n!$ rekursiv ist. In diesem Fall ist die iterative Programmvariante vorzuziehen, da sie einen geringeren Speicherplatzbedarf hat (vgl. Abschnitt 3.1).

Man sollte grundsätzlich auf die Verwendung der Rekursion verzichten, wenn man ohne Schwierigkeiten auch eine iterative Lösung finden kann.

Das eingangs aufgeführte Zitat darf also nicht so verstanden werden, dass auf jeden Fall die rekursive Variante die bessere ist!

Bei beiden Versionen wird das Problem in Teilprobleme zerlegt: Während in der rekursiven Implementierung von der komplizierteren zur einfachen Variante hin gearbeitet wird, verfährt das iterative Programm genau umgekehrt: Hier wird mit der einfachsten Stufe begonnen und sukzessive die nächst höhere berechnet, bis das Gesamtproblem gelöst ist. Diese Methode wird auch als „Bottom-Up-Methode“ bezeichnet. Die von oben nach unten gehende Vorgehensweise eines rekursiven Algorithmus nennt man entsprechend „Top-Down-Methode“.

Die Vorgehensweise bei der rekursiven Problemlösung entspricht der Top-Down-Methode. Die Vorgehensweise bei der iterativen Problemlösung entspricht der Bottom-Up-Methode.

Eine kompliziertere Rekursionsrelation ist die Berechnung der Fibonacci-Zahlen, die folgendermaßen definiert sind (Vgl. Abschnitt 1.3.5):

$$fib(0) = 0;\ fib(1) = 1;$$

$$fib(n) = fib(n\text{-}1) + fib(n\text{-}2) \quad \text{für } n > 1$$

Die rekursive Funktions-Prozedur könnte folgendermaßen aussehen:

```
function Fib(n: integer): LongInt;
begin
  if n=0 then Fib:=0
         else
           IF n=1 then Fib:=1
                  else Fib:= Fib(n-1) + Fib(n-2);
end;
```

Jeder rekursive Aufruf der Funktions-Prozedur mit n > 1 führt zu zwei weiteren Aufrufen, wodurch die Gesamtzahl der Aufrufe exponentiell wächst. In diesem Algorithmus werden Teillösungen mehrfach berechnet: Ist beispielsweise Fib(5) gesucht, so wird durch den mehrfach-rekursiven Aufruf Fib := Fib(4) + Fib(3) letztlich Fib(3) zweimal berechnet, da Fib(4) selbst wiederum Fib(3) aufruft. Aus demselben Grund wird Fib(2) doppelt berechnet.

Insgesamt wird durch den Aufruf Fib(5) letztlich 15 mal die Funktion Fib aufgerufen. Diesen Sachverhalt verdeutlicht der folgende Baum:

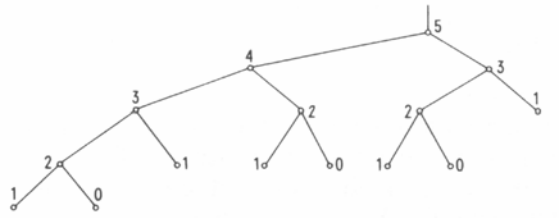


Abb. 12: Die 15 Aufrufe von Fib(5) [61]

Das vorgestellte rekursive Programm ist zwar elegant und leicht zu finden, aber es arbeitet sehr ineffizient; die vielen Mehrfachberechnungen verlängern die Laufzeit erheblich und benötigen viel Speicherplatz.

Die wiederholte Berechnung der gleichen Werte kann bei einer iterativen Implementierung unter Verwendung von Hilfsvariablen vermieden werden. Auch hier ist die iterative Programmvariante der rekursiven vorzuziehen, obwohl es schwieriger ist, die iterative Lösung zu finden.

[61] Abb. aus Wirth 154.

Wählen wir Hilfsvariablen x und y, für die die Beziehung $x = \text{Fib}(i)$ und $y = \text{Fib}(i\text{-}1)$ gilt, und z eine weitere Hilfsvariable zur Zwischenspeicherung, dann kann die Berechnung von Fib(n) folgendermaßen iterativ implementiert werden (für $n > 0$):

```
function Fib(n: integer): LongInt;
var i, x, y, z: LongInt;
begin
  x:=1; y:=0;
  for i:=1 to n do
    begin
      z:=x; x:=x+y; y:=z;
    end;
  Fib := x ;
end ;
```

Interessant ist, dass in der Programmiersprache PROLOG (siehe Abschnitt 2.3), die über keine Schleifenkonstrukte verfügt, das Problem so gelöst wird, dass neu berechnete Werte sogleich in der Wissensbasis gespeichert werden. Damit spart man wiederholtes Berechnen und ersetzt es durch ein Nachschlagen in der Wissensbasis.[62] Dies ist ein Beispiel für ein selbstlernendes System, für einen Schritt auf dem Weg zur Schaffung künstlicher Intelligenz (Wahlthema in Jahrgangsstufe 13/I).

Wie bereits in Abschnitt 2.3 aufgeführt, kann jedes rekursive Programm in ein rein iteratives umgeformt werden. Oft gelingt es jedoch nicht, das iterative Programm zu finden, vor allem bei Algorithmen, die eher rekursiv als iterativ sind.

Erinnern wir uns an das in Abschnitt 2.2 vorgestellte Problem „Die Türme von Hanoi“. Die unter dem Stichpunkt „Ein Lösungsweg für Menschen“ aufgeführte Lösungsvariante zeigt, dass auch ein iterativer Lösungsweg grundsätzlich möglich ist.

Während das rekursive Programm viel Speicherplatz braucht, um die vielen angefangenen Ausführungen der rekursiven Prozedur zwischenzuspeichern, würde ein iteratives Programm auf der Grundlage des vorgestellten Verfahrens, das immer dieselben Schritte innerhalb einer einfachen Schleife wiederholt, kaum Speicherplatz benötigen. Allerdings ist die Implementierung dieser Lösung ausgesprochen schwierig,

[62] Röhner (Informatik mit Prolog) 46.

sie ist in keinem gängigen Informatikbuch zu finden. Hier ist die rekursive Lösung die weitaus naheliegendere, elegantere und kürzere.

Auf Rekursion sollte man immer dann verzichten, wenn eine iterative Lösung offensichtlich ist. Lässt sich im Gegensatz zur Iteration ein rekursiver Algorithmus leicht auffinden und elegant formulieren, so ist die Rekursion vorzuziehen.[63]

Wie in Abschnitt 2.3 dargestellt wurde, erlauben manche Programmiersprachen keine Iteration, so dass immer die rekursive Umsetzung gewählt werden muss. Solche Sprachen setzen häufig zur Optimierung primitive Rekursionen intern als Iterationen um.[64]

[63] Siehe Wehrheim 19.

[64] Aus: http://de.wikipedia.org/wiki/Rekursion.

4 TYPISCHE ANWENDUNGEN

Es gibt viele Probleme, die sich mittels einer rekursiven Algorithmisierung deutlich leichter lösen lassen, als mit einer iterativen. Im folgenden werden einige typische Beispiele dazu vorgestellt. Zugleich soll die rekursive Denkweise durch das Aufgreifen in verschiedenen Zusammenhängen weiter veranschaulicht und geübt werden.

4.1 Listen

Ein Liste kann zweckmäßig rekursiv definiert werden: Eine Liste mit Grundtyp T ist entweder

1. die leere Liste oder
2. die Verkettung eines Elementes vom Typ T mit einer Liste vom Grundtyp T.[65]

Es handelt sich um eine Art "strukturelle" Rekursion. Daher bietet es sich an, Operationen auf Listen ebenfalls rekursiv zu definieren.

„Wenn rekursive Datenstrukturen in herkömmlichen Lehrbüchern eingeführt werden, sind die Algorithmen für die Manipulation dieser Datenstrukturen oft in einer iterativen Form angegeben. Häufig wird damit die Symmetrie und Einfachheit geopfert, die mit einer rekursiven Lösung möglich ist. Wenn eine rekursive Datenstruktur verwendet wird, so liefern die dieser Struktur zugrundeliegenden Relationen ein natürliches Modell für den für den rekursiven Algorithmus, mit dem diese Daten bearbeitet werden können.“[66]

Eine typische Art und Weise, rekursiv mit Listen zu arbeiten, ist, den Kopf zu bearbeiten und dann mit dem Rest in die Rekursion zu gehen.

„Listenstrukturen sind ein geeignetes Beispiel, um die Denkweise der linearen Rekursion zu veranschaulichen und einzuüben.“[67]

---

[65] Wirth 219.

[66] Roberts 167.

[67] Damann 223.

Im Folgenden werden die Implementierungen zweier ausgewählter Operationen auf verketteten Listen vorgestellt, wobei ausschließlich die Rekursion als Kontrollstruktur benutzt wird.

```
type  List = ^ListRecord;
      Element = ^ElementRecord;
      ListRecord = record
            first: Element
            end;
      ElementRecord = record
            content: integer;
            next: Element;
            list: List
            end;


var   l1, l2: List;
      e: Element;
      i: integer;
      test: boolean;


function Length(l: List): integer;
(* liefert die Anzahl der Elemente einer Liste *)

   function LengthElements(e: Element): integer;
   begin
     if e = nil
       then LengthElements := 0
       else LengthElements := 1 + LengthElements(e^.next)
   end;

begin
  Length := LengthElements(l^.first)
end;


function Find(l: List; i: integer): Element;
(* sucht das erste Element mit einem gegebenen Wert *)

  function FindElement(e: Element; i: integer): Element;
  begin
    if e = nil
      then FindElement := nil
      else
        if e^.content = i then FindElement := e
                          else FindElement := FindElement(e^.next, i)
  end;

begin
  Find := FindElement(l^.first, i)
end; [68]
```

[68] Prog. aus: http://www.informatik.fh-muenchen.de/~schieder/programmieren-2-ss96/rekursion.html.

Wie man sieht, lässt sich die Verarbeitung verketteter Listen sehr elegant mit rekursiven Funktionen abwickeln.

Bei der eben betrachteten Datenstruktur „Verkettete Liste“ handelte es sich um eine lineare Struktur. Rekursion kann aber auch zur Definition viel komplizierterer Strukturen verwendet werden, z. B. von Baumstrukturen.

4.2 Binäre Bäume

Eine Baumstruktur vom Grundtyp T ist entweder

1. die leere Struktur oder
2. ein Knoten vom Typ T mit einer endlichen Zahl verknüpfter, voneinander verschiedener Baumstrukturen mit Grundtyp T, so genannter *Teilbäume*.[69],[70]

Die Struktur „Binärer Baum“ als Sonderfall eines Baums ist demnach ebenso rekursiv definiert: „Binäre Bäume“ sind entweder leer oder bestehen aus der Wurzel und einem linken und einem rechten binären Teilbaum.

Die Anwendung der Rekursion beim Durchwandern oder Löschen eines Binären Baums gestaltet sich daher als sehr elegant. Sie ist leicht verständlich und dadurch eher fehlerfrei.

Beispiel: Durchlaufen eines binären Baums

```
procedure PrintTreeInOrder(tree: BinaryTree;)
begin
    if tree^.left <> nil then PrintTreeInOrder(tree^.left);
    write('[',tree^.data,']');
    if tree^.right <> nil then PrintTreeInOrder(tree^.right);
end; (* InOrder *)
```

Die Realisierung mit einem iterativen Programm ist viel aufwändiger und länger. Bei Bäumen ist die rekursive Lösung meist leichter zu verstehen als die iterative.

---

[69] Wirth 219.

[70] Aus den beiden Definitionen von Liste und Baum ergibt sich, dass die Liste eine Baumstruktur ist, in der jeder Knoten höchstens einen Teilbaum besitzt.

Ein weiteres Beispiel bezieht sich auf einen „Binären Suchbaum“, der die Eigenschaft hat, dass für jeden Knoten gilt:

- Informationen in allen Knoten des linken Unterbaumes sind kleiner
- Informationen in allen Knoten des rechten Unterbaumes sind größer

Dafür muss vorausgesetzt werden:

- Informationen müssen nach irgendeinem Kriterium vergleichbar sein
- Informationen kommen höchstens einmal im Baum vor

Als Beispiel dient hier ein Binärer Suchbaum mit Primzahlen zwischen 20 und 40:

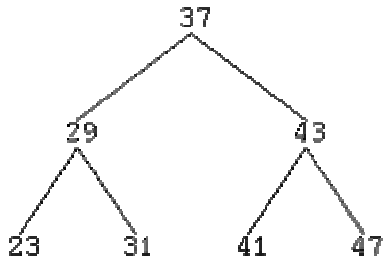


Abb. 13: Binärer Suchbaum mit Primzahlen zwischen 20 und 40

Da die Informationen im Baum organisiert sind, kann man effizient nach der Information *x* mit folgendem Verfahren suchen:

- Knoten leer: Information nicht gefunden
- Knoten enthält *x*: gefunden
- Information im Knoten kleiner als *x*: im rechten Teilbaum weitersuchen
- Information im Knoten größer als *x*: im linken Teilbaum weitersuchen

Beispiel: Suche nach Information "41"

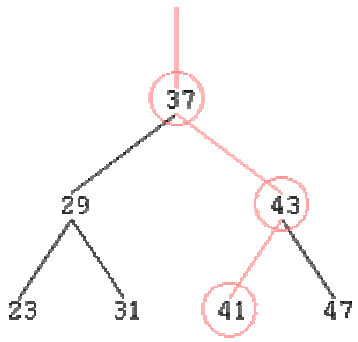


Abb. 14: Suche nach Information 41

Die folgende Funktion sucht den Knoten mit der Information *x* im Baum mit der Wurzel *n* und liefert den gefundenen Knoten zurück. Wenn *x* nicht im Baum vorkommt, wird ein leerer Knoten zurückgeliefert.

```
function Find(n: Node; x: Info): Node;
begin
  if Empty(n)
    then Find := n
    else if Info(n) = x
           then Find := n
            else if Info(n) < x
                then Find := Find(Right(n), x)
                else (* Info(n) > x *)
                   Find := Find(Left(n), x)
end;
```
[71]

[71] Abbildungen und Programm aus: http://www.informatik.fh-muenchen.de/~schieder/programmieren-2-ss96/baeume.html#suche-0.

4.3 Das Prinzip „Teile und Herrsche“

Die Rekursion ist ein wesentlicher Bestandteil einiger Entwurfsstrategien für effiziente Algorithmen, insbesondere der so genannten „Teile und Herrsche“-Strategie.

Das Prinzip "Teile und Herrsche“ ist auch als “divide et impera“ (lateinisches Original) oder "divide and conquer" (englische Übersetzung) bekannt. Der Name stammt von einem erfolgreichen Prinzip der römischen Machtpolitik[72]. In der Informatik handelt es sich um ein „wichtiges Prinzip der Softwareentwicklung, bei dem eine Gesamtproblemstellung an möglichst gut gewählten Schnittstellen in kleinere Teile zerlegt wird. Die Aufteilung lässt sich TopDown beliebig fortsetzen, solange bis handhabbare (überschaubare, delegierbare, spezifizierbare, direkt implementierbare, ...) Aufgaben entstanden sind.“ [73]

Die Aufteilung in kleinere Probleme und deren Behandlung geschieht rekursiv bis hinab zum trivialen Fall. In Abschnitt 2.2.2 haben wir zu diesem Verfahren bereits ein Beispiel kennen gelernt: Die Türme von Hanoi.

„Probleme, die ihrer Natur nach zur Strategie von ‚Teile und Herrsche’ passen, lassen sich besonders gut rekursiv lösen. Damit ein Problem ein Kandidat für eine rekursive Lösung nach dem Prinzip ‚Teile und Herrsche’ ist, muss es folgende Eigenschaften besitzen:

1. Das Originalproblem muss sich in einfachere Varianten von sich selbst zerlegen lassen.
2. Die Zerlegung in Teilprobleme muss letztendlich auf so einfache Varianten des Originalproblems führen, dass sie ohne weitere Zerlegung gelöst werden können.
3. Wenn alle Teilprobleme gelöst sind, müssen die Lösungen so zusammengesetzt werden können, dass sich eine Lösung des Originalproblems ergibt.“[74]

---

[72] Dieser Begriff wurde schon von C. Julius Caesar geprägt, der die Divide-and- Conquer Strategie im Krieg gegen die Gallier einsetzte ("de bello gallico"). Er nützte die Teilung Galliens in verschiedene Stämme aus. Diese waren sich dann untereinander so uneinig, dass sie sich zu keiner Gesamtmacht gegen die römischen Truppen zusammenschließen konnten. Caesar hatte es also nicht mit einem recht großen Problem, sondern nur mit mehreren lösbaren Teilproblemen zu tun.
(aus: http://www.amigascene.com/deutsch/0x009B.html)

[73] Aus: http://www.wikiservice.at/dse/wiki.cgi?DivideEtImpera.

[74] Wehrheim 7.

Das Prinzip „Teile und Herrsche“ wird in zahlreichen Algorithmen angewendet, beispielsweise bei den Sortierverfahren *Mergesort* und *Quicksort*.

An dieser Stelle möchte ich den einfacheren Algorithmus des *MergeSort* vorstellen, der durchaus im Grundkurs behandelt werden kann. Zum *Quicksort*, der im Durchschnitt um einen konstanten Faktor schneller als der *Mergesort* ist, findet sich ein Arbeitsblatt mit Lösung im Anhang. Dieser ist komplizierter und daher eher dem Leistungskurs vorbehalten.

4.3.1 Mergesort

Nach der Idee „Teile und Herrsche“ wird die zu sortierende Folge zunächst in zwei Hälften geteilt („Teile“), die jeweils für sich sortiert werden („Herrsche“). Dann werden die sortierten Hälften zu einer insgesamt sortierten Folge verschmolzen[75].

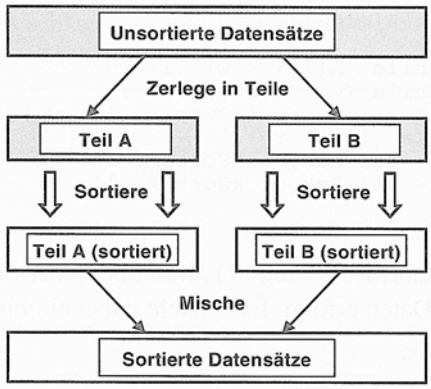


Abb. 15: Prinzip von MergeSort[76]

Die folgende Prozedur *mergesort* sortiert ein Feld *Sortierfeld* vom unteren Index *Anfang* bis zum oberen Index *Ende*.

[75] engl.: to merge = verschmelzen

[76] Abb. aus Gumm 294.

```
procedure mergesort(var Sortierfeld: TableArray;
                        Anfang, Ende: integer);
var Mitte: integer;
begin
  if Anfang < Ende then begin
      Mitte:=(Anfang + Ende)div 2;
      mergesort(Sortierfeld, Anfang, Ende);
      mergesort(Sortierfeld, Mitte+1, Ende);
      merge(Sortierfeld, Anfang, Ende);
    end;
end;
```

Zunächst wird die Mitte zwischen *Anfang* und *Ende* bestimmt. Dann wird die untere Hälfte der Folge (von *Anfang* bis *Mitte*) sortiert und anschließend die obere Hälfte (von *Mitte+1* bis *Ende*), und zwar durch rekursiven Aufruf von *mergesort*. Die Rekursion endet, wenn *Anfang* gleich *Ende* wird, d. h. die dann zu sortierende Teilfolge besteht nur noch aus einem Element.

Anschließend werden die sortierten Hälften durch Aufruf der Prozedur *merge* miteinander verschmolzen. Dazu werden zunächst die beiden Hälften hintereinander in ein neues Feld *Hilfsfeld* kopiert. Dann werden die beiden Hälften mit einem Index *i* und einem Index *j* elementweise verglichen und jeweils das nächstgrößte Element zurück nach *Sortierfeld* kopiert.

Dieses Konzept ist zwar recht einfach, doch ist es schwierig zu implementieren. Denn es müssen abwechselnd von beiden Hälften Elemente kopiert werden, wobei man feststellen muss, wann eine Hälfte fertig kopiert ist. Dabei ist nicht bekannt, um welche Hälfte es sich handelt. Dann muss noch der Rest der anderen Hälfte in das Hilfsfeld kopiert werden.

Die Implementierung wird einfacher, wenn man die zweite Hälfte in umgekehrter Reihenfolge in das Hilfsfeld kopiert. Die Prozedur *merge* durchläuft dann mit dem Index *i* die erste Hälfte von links nach rechts und mit dem Index *j* die zweite Hälfte von rechts nach links. Das jeweils nächstgrößte Feldelement wird in das Feld *Sortierfeld* zurückkopiert. Wenn *i* und *j* sich überkreuzen, d. h. wenn $i > j$ wird, ist das gesamte Feld kopiert worden. (Es ist nicht notwendig, dass *i* und *j* in "ihren" Hälften bleiben.)[77]

[77] Vgl. http://www.delphi-source.de/tipps/algorithmen/?id=10 (von Hans-Werner Lang)

Die entsprechende Implementierung der Prozedur *merge* sieht folgendermaßen aus:

```
procedure merge(var Sortierfeld: TableArray; Anfang, Ende: integer);
var i, j, k, Mitte, n: integer;
    Hilfsfeld: TableArray;
begin
  n:=Ende-Anfang+1;
  k:=0;
  Mitte:=(Anfang+Ende)div 2;
  // untere Hälfte in das Hilfsfeld kopieren
  i:=0;
  while i<=Mitte do begin
    Hilfsfeld[k]:=Sortierfeld[i];
    k:=k+1; i:=i+1;
  end;
  // obere Hälfte in umgekehrter Reihenfolge in das Hilfsfeld kopieren
  j:=Ende;
  while j>=Mitte+1 do begin
    Hilfsfeld[k]:=Sortierfeld[j];
    k:=k+1; j:=j-1;
  end;
  i:=0; j:=n-1; k:=Anfang;

  // jeweils das nächstgrößere Element zurückkopieren
  // bis i und j sich überkreuzen
  while i<=j do begin
    if Hilfsfeld[i]<=Hilfsfeld[j] then begin
      Sortierfeld[k]:=Hilfsfeld[i];
      k:=k+1;
      i:=i+1;
    end
    else begin
      Sortierfeld[k]:=Hilfsfeld[j];
      k:=k+1;
      j:=j-1;
    end;
  end;
end;
```

Ein Nachteil des Mergesort ist der zusätzliche Speicherplatzbedarf für das temporäre Feld *Hilfsfeld*.

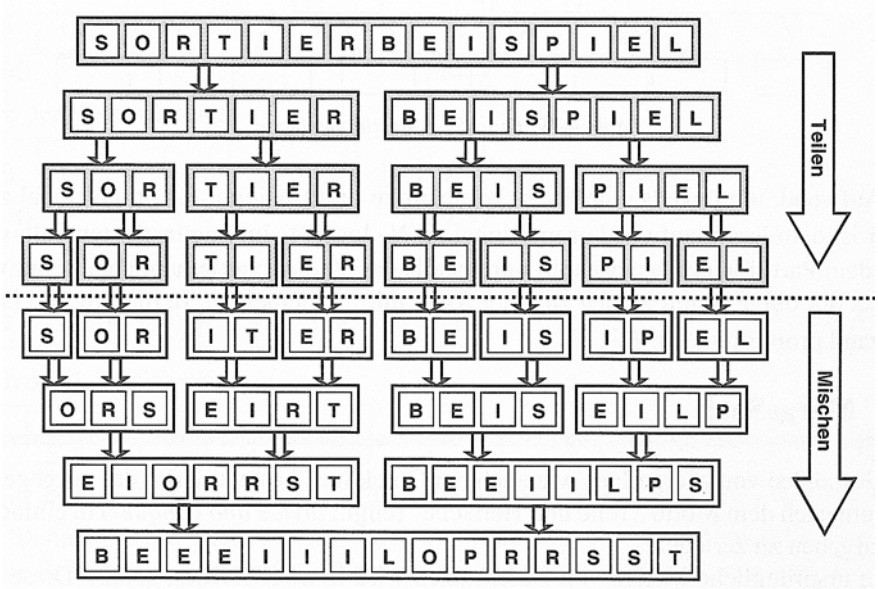


Abb. 16: MergeSort bei einem Beispiel[78]

[78] Abb. aus Gumm 294.

4.4 Backtracking

Wenn sich zu einem Problem ein Lösungsalgorithmus nicht direkt angeben lässt, kann man ihn höchstens durch systematisches Ausprobieren finden. Dies erfolgt durch die Methode *Versuchen und Nachprüfen* (= Versuch und Irrtum / trial and error). Die Vorgehensweise ist einfach: Man sucht systematisch gemäß einer vorgegebenen Reihenfolge Schritte Richtung Ziel und zeichnet sie auf. Stellt sich später heraus, dass sie in eine Sackgasse führten, so macht man sie wieder rückgängig und löscht die Aufzeichnungen. Diese Strategie heißt *backtracking*. Hierbei spielt wiederum die Rekursion eine wichtige Rolle. Der Prozess des Versuchens und Nachprüfens wird gewöhnlich in einzelne Teilschritte zerlegt, die sich auf natürliche Art in rekursiver Form ausdrücken lassen und aus der Untersuchung einer endlichen Zahl untergeordneter Schritte bestehen.[79]

Im Folgenden werde ich ein typisches Backtracking-Problem ausführlich darlegen, das *Problem der 8 Damen* auf dem Schachbrett. Weitere Beispiele für diese Strategie sind das *Irrgarten-Problem* und das *Travelling-Salesperson-Problem (TSP)*, über die Informations- und Arbeitsblätter sowie Delphi-Programme im Anhang bereitgestellt sind. Zur Lösung des *Springer*[80]- und *Rucksack-Problems* finden sich im Anhang ebenfalls Delphi-Programme.

4.4.1 Das Problem der 8 Damen

Der berühmte Mathematiker C. F. Gauß hat sich bereits 1850 mit folgendem Problem auseinandergesetzt:

Acht Damen sind auf einem Schachbrett so aufzustellen, dass keine Dame eine andere bedroht.

Gauß konnte dieses Problem nicht vollständig lösen, da es die charakteristische Eigenschaft hat, dass es sich nicht analytisch lösen lässt.[81] Erst mit Hilfe eines schnellen Computers, der dem Menschen die Arbeit des systematischen Probierens abnimmt, können solche Lösungen leicht gefunden werden.[82]

---

[79] Siehe Wirth 162.

[80] Die Lösung des Springer-Problems führte vor einiger Zeit ein Neunjähriger als Wette bei „Wetten dass?“ vor. Die Wette gibt’s als Video im Internet und eignet sich gut zum Einstieg in das Problem. (Rösselsprungwette: 22.02.2003 http://www.zdf.de/ZDFde/inhalt/3/0,1872,2035203,00.html)

[81] Siehe Wirth 168.

[82] Siehe Wehrheim 35.

Oft wird dieses Problem allgemeiner als „N-Damenproblem“ formuliert und bezieht sich dann auf N Damen und ein N×N-Schachbrett. Eine weitere Erweiterung des Problems besteht darin, dass nicht nur eine Lösung, sondern alle Lösungen des Problems gesucht sind.

Die Damen schlagen (und bedrohen) nach den üblichen Schachregeln waagrecht, senkrecht und diagonal.

Demnach ergeben sich z. B. für das 4-Damenproblem zwei Lösungen:

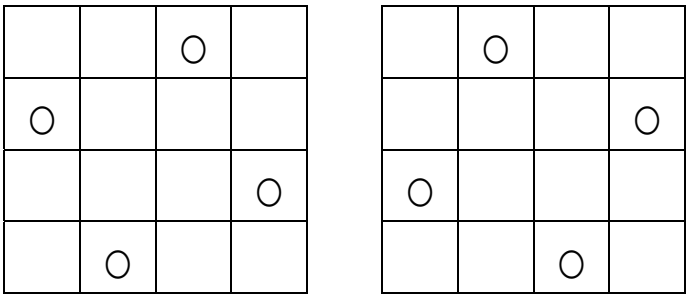


Abb. 17: Lösungen für das 4-Damen-Problem

Durch systematisches Probieren kann man alle Lösungen finden.

Nach den genannten Regeln ist klar, dass nicht mehr als eine Dame pro Spalte, Zeile und Diagonale stehen darf. Der Ansatz für den Algorithmus ist nun, dass man das Schachbrett spaltenweise von links nach rechts durchgeht und in jede Spalte eine Dame zu platzieren versucht, indem in jeder Spalte alle Positionen durchprobiert werden, in denen die Dame „kollisionsfrei“ zu allen weiter links platzierten Damen passt. Wenn in einer Spalte keine Dame mehr gesetzt werden kann oder alle Damen gesetzt sind, bricht man ab.

Man stellt also zunächst die erste Dame auf das erste Feld in der ersten Spalte. Da dann in dieser Spalte keine Dame mehr stehen darf, prüft man der Reihe nach von oben nach unten die Felder der zweiten Spalte. Das dritte Feld der zweiten Spalte ist nicht bedroht; dort stellt man die zweite Dame ab. Zum Aufstellen der dritten Dame werden alle Felder der dritten Spalte geprüft. Doch es stellt sich heraus, dass alle bedroht sind:

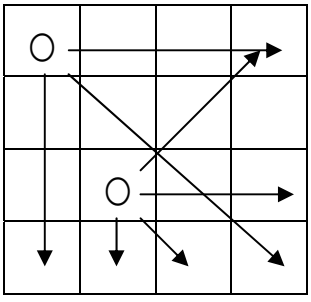


Abb. 18: Bedrohte Felder durch die ersten beiden gesetzten Damen

Die dritte Dame kann also nicht gesetzt werden. Daher wird nun die zuletzt aufgestellte Dame in der zweiten Spalte zurückgenommen und auf das nächste Feld in dieser Spalte, also auf das letzte, gestellt. Dann kann für die dritte Dame ein unbedrohter Platz in der dritten Spalte gefunden werden:

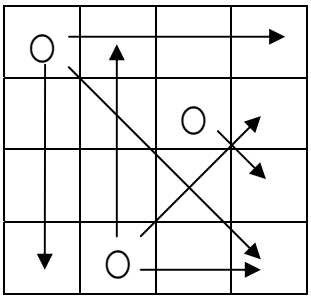


Abb. 19: Bedrohte Felder durch die ersten drei gesetzten Damen

Jetzt kann aber in die vierte Spalte keine Dame mehr gestellt werden. Daher setzt hier erneut das Backtracking ein: Die Aufstellung der dritten Dame wird zurückgenommen. Aber ihre Aufstellung auf eines der nächsten Felder in der dritten Spalte ist nicht erlaubt. Also muss in weiteren Backtracking-Schritten auch die zweite und schließlich die erste Dame zurückgenommen werden. Die erste Dame wird dann auf das zweite Feld in der ersten Spalte gestellt usw.

Implementierung mit rekursiven Funktionen[83]

Die Verwaltung der Damen geschieht in einem Feld für die acht Spalten: `Damen[1..8] of integer`. Jeder Eintrag `Damen[i]` gibt die Zeile an, in der die Dame in Spalte i steht.

Umgangssprachlich kann man den Lösungsalgorithmus wie folgt formulieren:

- Starte mit der ersten Dame in der ersten Spalte.
- Mache hier für jede Zeile folgendes:
  - Setze die Dame dorthin.
  - Probiere nun, die restlichen Damen in den übrigen Spalten zu setzen.
  - Falls es geklappt hat,
    dann haben wir eine Lösung und sind fertig
    sonst nimm die Dame wieder vom Schachbrett.

Für das Problem „Probiere nun, die restlichen Damen in den übrigen Spalten zu setzen“ kann man genau den gleichen Lösungsalgorithmus verwenden – mit dem Unterschied, dass man nun mit der zweiten Dame in der zweiten Spalte anfängt. Das gleiche gilt

[83] Beschreibung und Programm von D. Garmann, Gymnasium Odenthal.

schließlich auch für die 3., 4. ... und 8. Dame. Wir erhalten so den rekursiven Backtrackingalgorithmus:

Algortihmus „*Versuche_Dame_zu_setzen*“

- Für Zeile von 1 bis 8 tue folgendes:
  - Falls NICHT *Dame_bedroht(Spalte, Zeile)*
    dann
    - Damen[Spalte] ← Zeile
    - *Setze_Bedrohung(Spalte, Zeile)*
    - Falls Spalte = 8
      dann
      - Zeige die Lösung an

      sonst
      - ***Versuche_Dame_zu_setzen (Spalte + 1, Damen)***
    - *Lösche_Bedrohung(Spalte, Zeile)*
    - Damen[Spalte] ← 0

Die Frage, ob *Dame_bedroht(Spalte, Zeile)* ist und die Routinen für *Setze_Bedrohung(Spalte, Zeile)* und *Lösche_Bedrohung(Spalte, Zeile)* werden im Folgenden konkretisiert:

Eine Dame bedroht die Spalte und die Zeile, in der sie steht. Da die Datenstruktur so gewählt wurde, dass in jedem Fall nur maximal eine Dame in einer Spalte gesetzt ist, erübrigt sich die Frage, ob eine neu zu setzende Dame durch eine andere Dame in der gleichen Spalte bedroht sein könnte. Anders verhält es sich mit der Zeile. Hierfür wird ein Feld angelegt, in dem festgehalten wird, welche Zeilen bereits von Damen besetzt sind und somit andere Damen in diesen Zeilen bedrohen würden: `Zeile_bedroht[1..8] of Boolean.`

Neben der Zeile und Spalte werden durch eine Dame auch die Diagonalen (Aufwärtsdiagonale und Abwärtsdiagonale) bedroht. Schauen wir uns dazu zwei Beispiele an:

	1	2	3	4	5	6	7	8
1								
2								
3								O
4							O	
5						O		
6					O			
7				O				
8			O					

	1	2	3	4	5	6	7	8
1			O					
2				O				
3					O			
4						O		
5							O	
6								O
7								
8								

Im ersten Beispiel liegen alle Damen auf einer Aufwärtsdiagonalen – bedrohen sich also gegenseitig. Die Besonderheit hieran ist, dass für jede Dame gilt, das die Summe der Spalte und Zeile, in der sie steht, konstant ist. In diesem Fall ist Spalte + Zeile = 11, d. h. die Aufwärtsdiagonale Nr. 11 wird von den eingetragenen Damen belegt. Insgesamt gibt es 15 Aufwärtsdiagonalen, deren eventuelle Bedrohung durch eine Dame festgehalten werden muss. Dazu wählen wir ein Feld `AufDiag_bedroht[2..16] of Boolean,` in dem beim Setzen einer Dame das Feldelement mit dem Index *Spalte+Zeile* auf *true* gesetzt wird. Die Nummerierung ergibt sich daraus, dass die Summe aus Spalte und Zeile mindestens 2 (1+1) und höchstens 16 (8+8) ergibt.

Im zweiten Beispiel liegen alle Damen auf einer Abwärtsdiagonalen Die Besonderheit an dieser Situation ist, dass für jede Dame gilt, das die Differenz der Spalte und Zeile, in der sie steht, konstant ist. In diesem Fall ist Spalte – Zeile = 2, d. h. die Abwärtsdiagonale Nr. 2 wird von den eingetragenen Damen belegt. Insgesamt gibt es auch hier 15 Abwärtsdiagonalen, deren Bedrohung durch eine Dame festgehalten werden muss. Wählen wir auch hier wieder ein Feld mit 15 Einträgen, so sollten wir die Nummerierung umstellen. Die Differenz „Spalte – Zeile“ nimmt mindestens den Wert –7 (1–8) und höchstens den Wert +7 (8–1) an. Daraus ergibt sich die Feldvariable `AbDiag_bedroht[-7..7] of Boolean,` in der beim Setzen einer Dame das Feldelement mit dem Index *Spalte–Zeile* auf *true* gesetzt wird.

Der Algorithmus für das Setzen und Löschen einer Bedrohung ist damit relativ einfach:

Algorithmus *„Setze_Bedrohung“*	***„Lösche_Bedrohung“***
• Zeile_bedroht[Zeile] ← true	..... ← false
• AufDiag_bedroht [Spalte + Zeile] ← true	..... ← false
• AbDiag_bedroht [Spalte – Zeile] ← true	..... ← false

Der Algorithmus zur Feststellung, ob *Dame_bedroht(Spalte, Zeile)* ist, ist dann nur noch eine Formsache:

Algorithmus *„Dame_bedroht“*

• Dame_bedroht ← Zeile_bedroht[Zeile] ODER
AufDiag_bedroht [Spalte + Zeile] ODER
AbDiag_bedroht [Spalte – Zeile]

Die Implementierung der Prozeduren in Delphi

Zum Start wird die Prozedur *versuche(1, N)* aufgerufen, wobei die „1“ für den Beginn mit der 1. Spalte steht und das „N“ für die Anzahl der zu setzenden Damen.

```
const max_n  = 10;
      max_n2 = 20;
      max_n3 =  9;

var          {global genutzte Variablen}
      anz : Integer;
      dame: array [1..max_n] of Byte;
      Inf_zeile : array [1..max_n]  of Boolean;
      Inf_aufdia: array [2..max_n2] of Boolean;
      Inf_abwdia: array [-max_n3..max_n3] of Boolean;


  procedure init_Infos;
  var i: Integer;
  begin
     anz:= 0;
     for i:=1 to max_n  do dame[i]:= 0;
     for i:=1 to max_n  do Inf_zeile [i]      := false;
     for i:=2 to max_n2 do Inf_aufdia[i]      := false;
     for i:=-max_n3 to max_n3 do Inf_abwdia[i]:= false;
  end;

  function bedroht (i,k: Byte): Boolean;
  begin
     bedroht:= Inf_zeile[k] or Inf_aufdia[i+k] or Inf_abwdia[i-k]
  end;

  procedure setze_Bedrohung (i,k: Byte; wert: Boolean);
  begin
     Inf_zeile[k]:= wert;
     Inf_aufdia[i+k]:= wert;
     Inf_abwdia[i-k]:= wert
  end;

  procedure versuche (sp, max: Byte);
  var ze: Byte;
  begin
     with F_nDamen do
     for ze:= 1 to max do
        if not bedroht (sp, ze) then
        begin
          dame[sp]:=ze;
          setze_Bedrohung (sp, ze, true);
          if sp = max then inc(anz)
                      else versuche (sp+1, max);
          setze_Bedrohung (sp, ze, false);  {Bedrohung löschen}
         dame[sp]:=0;
       end;
   end;
```

Das komplette Delphi-Programm, das noch um eine schöne Ausgabe erweitert ist, findet sich im Anhang.

5 RESÜMEE

> *Von all den Ideen, die ich Kindern vorgestellt habe, zeichnet sich die Rekursion als die eine Idee aus, die eine besonders aufgeregte Reaktion hervorrufen konnte. Ich glaube, das kommt zum Teil daher, dass der Gedanke einer endlosen Fortsetzung die Phantasie jedes Kindes anspricht, und zum Teil daher, dass die Rekursion selbst Wurzeln in der Alltagskultur hat.*
>
> *Seymour Papert: Gedankenblitze*

Wir haben nun die vielen interessanten Facetten der Rekursion kennen gelernt und gesehen, wie mit Hilfe rekursiver Programmiertechnik auch komplexe Aufgabenstellungen durch erstaunlich kurze und einfache Algorithmen programmtechnisch sauber gelöst werden können.

Die Rekursion stellt nicht nur für zahlreiche Einzelprobleme eine elegante Lösungsmöglichkeit dar, z. B. nach dem „Teile und Herrsche“-Prinzip. Typische Anwendungs*bereiche* für rekursive Prozeduren sind Backtracking-Verfahren und Algorithmen, die auf rekursiven Datentypen operieren.

Das Thema Rekursion übt für viele Schülerinnen und Schüler einen besonderen Reiz aus - die Denkweise ist ungewohnt, und die scheinbare Unendlichkeit verwundert und weckt Interesse. Die meisten der typischen Anwendungen für Rekursion sind interessante, leicht verständliche aber doch knifflige Problemstellungen, die faszinieren und dadurch motivieren. Außerdem können sie mit alltäglichen Gegenständen leicht nachgestellt werden. Dadurch haben die Schülerinnen und Schüler die Möglichkeit, sich erst einmal mit der Aufgabenstellung intensiv auseinandersetzen. Dabei reifen oft bereits Lösungsideen heran.

Ich denke, es lohnt sich daher, im Informatikunterricht dieses reizvolle Gebiet aufzugreifen. Es besteht sogar die Gefahr, dass über der Vielzahl interessanter Beispiele zur Rekursion das ganze Schulhalbjahr vergeht. Neben dem Erkennen und Anwenden von rekursiven Prozeduren für spezielle Problemlösungen sollen die Schülerinnen und Schüler auch lernen, den Einsatz von rekursiven oder iterativen Problemlösungen kritisch gegeneinander abzuwägen.

Dass Rekursion ein allgemeines Konstruktionsverfahren und von grundlegender Bedeutung in Mathematik, Informatik und, wie wir oben gesehen haben, auch Linguistik und Musik ist, sollte den Schülerinnen und Schülern anhand einiger Beispiele im fächerübergreifenden Zusammenhang bewusst gemacht werden. Das Prinzip der Rekursion kann prinzipiell in allen Schulstufen in den Unterricht einbezogen werden. In der Mittelstufe sollte man sich dabei auf die Vermittlung des Prinzips selbst beschränken. In der Oberstufe wird dann das Augenmerk auf den Problemlösecharakter der Rekursion gelegt und anhand von dem Durchlaufen von Binärbäumen, rekursiven Such- und Sortierverfahren und den Anfängen der künstlichen Intelligenz erläutert und geübt.

6 LITERATURVERZEICHNIS

Bähnisch U.: *Praktische Informatik mit Delphi (Band 2)*.
Berlin: Cornelsen 2001.

Baumann R.: *Rekursion: Beispiele aus der Kombinatorik.*
In: LOG IN 8 (1988), Heft 5/6, S. 58-63.

Baumann R.: *Informatik für die Sek. II, Band 2*.
Stuttgart: Klett, 1993.

Damann P. und Wemßen J.: *Objektorientierte Programmierung mit Delphi*.
Stuttgart: Klett, 2003.

Dewdney A. K.: *Computer-Kurzweil*.
In: Spektrum der Wissenschaft 8 (1985), Heft 1, S. 8-13.

Duden „Informatik“, hrsg. vom Lektorat des Bi-Wiss..-Verl.,
2., vollst. überarb. und erw. Aufl. , Mannheim: Dudenverl., 1993.

Engelmann, Dr. Lutz (Hrsg.): *Informatik bis zum Abitur*.
Berlin: paetec, 2002.

Gellert W. (Hrsg.): *Mathematik-Ratgeber für Lehrer, Schüler, Eltern u. zum Selbststudium*. Frankfurt/M.: Deutsch, 1988.

Grewendorf G. u. a.: *Sprachliches Wissen*.
Frankfurt: Suhrkamp, 1987.

Gumm, H.-P. und Sommer M.: *Einführung in die Informatik.*
4., überarb. Auflage. München; Wien: Oldenbourg, 2000.

Hafenbrak B.: *Rekursion - sinnvoll im Unterricht?*
In: COMAL im Unterricht. Weinheim: Deutscher Studien Verlag, 1988.

Hessisches Kultusministerium: *Lehrplan Informatik*,
Gymnasialer Bildungsgang, Jahrgangsstufe 11 bis 13. Wiesbaden: 2003.

Hofstadter D. R.: *Gödel, Escher, Bach, ein Endloses Geflochtenes Band*.
9. Auflage. München: dtv, 2003.

Knöß P.: *Rekursive Prozeduren – eine Unterrichtsreihe (Teil 1)*.
In: LOG IN 5 (1985), Heft 5/6, S. 83-86.

Kuhlmann Gregor: *Programmiersprache TURBO-PASCAL*.
Hamburg: rororo, 1990.

Morgenstern Christian: *Alle Galgenlieder*.
Neugesetzte Ausgabe nach der Ausgabe Berlin 1932. Wiesbaden: Matrix, 2004.

Papert S.: *Gedankenblitze*.
Rowohlt, Reinbek bei Hamburg, 1985.

Roberts E.: *Rekursiv programmieren*.
München: Oldenbourg-Verlag 1987.

Röhner G.: *Rekursive Grafiken*.
Material zum WBK Informatik / Wochenlehrgang Nr. 2 (Delphi), 2002.

Röhner G.: *Informatik mit Prolog*.
2. vollständig neu überarb. Auflage. Wiesbaden: 2002.
(Materialien zum Unterricht; Hrsg.: Hess. Landesinstitut für Pädagogik)

Wehrheim O.: *Rekursives Problemlösen*.
Lehrbrief PAS 8. Studiengemeinschaft Darmstadt, Pfungstadt: 1989.

Wirth, N.: *Algorithmen und Datenstrukturen*.
Stuttgart/Leipzig/Wiesbaden: Teubner, 2000.

Internet-Seiten:

http://www.amigascene.com/deutsch/0x009B.html

http://www.delphi-source.de/tipps/algorithmen/?id=10

http://www.fim.uni-linz.ac.at/lva/rus/Rekursion/Rekursion.htm

http://www.informatik.fh-muenchen.de/~schieder/programmieren-2-ss96/rekursion.html

http://www.informatik.fh-muenchen.de/~schieder/programmieren-2-ss96/baeume.html#suche-0

http://www.mathekiste.de/fibonacci/projfibonacci.htm

http://www.wikiservice.at/dse/wiki.cgi?DivideEtImpera

http://www.zdf.de/ZDFde/inhalt/3/0,1872,2035203,00.html

Viele der **im Anhang** aufgeführten Arbeitsblätter stammen von Herrn Daniel Garmann vom Gymnasium Odenthal.

http://www.gymnasium-odenthal.de (Download → Informatik)