Rekursive Algorithmen

Einleitung

Autor: Siegfried Thoß

n der Programmiersprache Delphi - eigentlich ja Object Pascal - können Prozeduren und Funktionen sich selbst aufrufen. Diese Programmiertechnik bezeichnet man als ***rekursives Programmieren***.

Interessant ist, wie mit dem Programm-Werkzeug ***Rekursive Programmiertechnik*** sehr komplexe Aufgabenstellungen durch erstaunlich kurze und einfache Algorithmen programm-technisch sauber gelöst werden können.

Unser Lösungsplan, nach dem der Computer die Daten verarbeiten soll, stellt für den Computer eine Art Rezept dar, wie er vorzugehen hat. Eine solche eindeutige Beschreibung eines Verfahrens heißt ***Algorithmus***. Ein Algorithmus beschreibt präzise, wie der Computer aus ***Eingabedaten*** schrittweise nach dem ihm übergebenen ***Verarbeitungs"rezept"*** die gewünschten ***Ausgabedaten*** zu erzeugen hat. (Eingabe => Verarbeitung => Ausgabe).

Rekursive Algorithmen sind dann also "Computerrezepte", die mit Varianten von sich selber beschrieben werden (siehe Schritt Rekursion). Rekursion ist daher nicht nur eine Programmiertechnik, sondern ein ***wichtiges algorithmisches Prinzip zur Problemlösung***.

Rekursives Denken - liebe Leserin, lieber Leser, Du hast es vielleicht schon festgestellt - ist eine Denkweise, die uns, um sie zu verstehen, um manche Stunde Nachtruhe bringen kann. Aber das sollte es uns wert sein, wenn wir dafür bei unserer Programmierung durch ganz einfache rekursive Methoden viele Tagstunden harte Arbeit am Programmiertisch einsparen können.
Solltest Du zwischendurch den ganzen Kram hinhauen wollen, weil Du es einfach nicht kapierst, dann kann ich Dich nur um Geduld bitten. Ich selber beiße mir heute noch die Zähne an der Rekursion aus. Mir geht's also auch nicht besser! Aber ich versuche es.

Übrigens: Hast Du etwa die bekannten russischen Puppen zu Hause, von denen man eine in die andere stecken kann? Matroschkas nannten wir sie.
Hier hast Du ein typisches Anschauungsmodell für Rekursion: Willst Du alle Puppen in der großen Matroschka aufheben, musst Du IMMER DER REIHE NACH jeweils die nächstkleinere Figur verstauen. Anders geht das nicht!!
Und umgekehrt ist es genau so!!
Eben hast Du ein rekursives "Rezept" abgearbeitet - zumindest gedanklich.

Ehe wir beginnen, noch ein Wort zu diesem Tutorial:

In meinem Englisch-Wörterbuch wird ***tutorial*** mit ***Übung unter Anleitung*** übersetzt. An diese deutsche Bedeutung will ich mich in diesem Tutorial halten.
Du wirst hier einiges finden, das Du selbst probieren kannst. Du wirst aber auch manchen Satz finden, den ich mit einem Augenzwinkern geschrieben habe. Ich meine, Lernen muss nicht immer eine todlangweilige Angelegenheit sein. Ich hoffe nur, es gelingt mir.
Ich bin kein Lehrer, und gleich gar kein Informatiklehrer, aber ich hoffe, dass ich Dir die Rekursion trotzdem etwas schmackhaft machen kann. Viel Spaß dabei.

Solltest Du durch einen guten Gedanken eine Sache aufhellen können, lass es mich wissen. Danke schon jetzt.

Anmerkung: Die Datei mit den Beispielen findest Du hier.

Modularisierung

Um eine größere Programmieraufgabe zu lösen, müssen wir uns - ***vor*** allen Überlegungen über Algorithmen, vielleicht gar noch rekursive - zuerst einmal über die Möglichkeiten klar werden, die uns zur Verfügung stehen, um unser "Riesen"problem erfolgreich lösen zu können.

In den Pascal-Grundlagen hier bei Delphi-Source las ich neulich: Die objektorientierte Programmierung "soll die unstrukturierte Programmierung (am Anfang ein begin, dann Befehl auf Befehl und am Ende ein end, dazwischen möglichst viele Sprünge mit goto) ablösen." - In einem Lehrgang über Turbo Pascal kurz nach der Wende erlebte ich einen Mitteilnehmer, der ein doch etwas längeres Pascal-Programm "in einem Ritt" aufschrieb, also wirklich hintereinander, nur mit den Leer- oder Trennungszeichen, wenn sie Pascal unbedingt haben wollte. Und, o Wunder!, das Programm machte sogar, was es sollte.
(Würdest Du in so einem Programm Fehler suchen wollen?).

Um weder den einen noch den anderen Programmstil anwenden zu müssen, haben wir uns an den modularen Programmierstil gewöhnt.

Das Modularisierungs-Prinzip "Teile und Herrsche"

Dazu gehört zunächst, eine Aufgabe (unser "Riesen"problem) in einfachere Teilaufgaben zu zerlegen. Die Lösung der Teilaufgaben wird in Form von Prozeduren und Funktionen programmiert. Die Definition geeigneter Datenstrukturen kann die Formulierung einer Lösung wesentlich erleichtern.

In dem beiliegenden Programm Hanoi habe ich zum Beispiel in der Unit uHanoiDef, der Unit, die die Definitionen enthält, einen Datentyp

TZiele = (zuPlatzB, zuPlatzC);

definiert. Der dient mir dazu, mit diesen zwei Werten in einer Prozedur jeweils ein bestimmtes Ziel festzulegen. Ich hätte da auch eine ganz allgemeine Formulierung ohne extra Datentyp verwenden können, aber so ist der Quelltext wesentlich einfacher zu lesen, für andere Programmierer und nach Wochen und Monaten auch für mich.

Wenn ich mich an so ein Modularisierungsprinzip halten will, kann ich dieses Zerlegen in Teilaufgaben auf zweierlei Art lösen:

1. Die Top-Down-Methode

Eine Programmentwicklung nach der Top-Down-Methode geht von der Leitfrage aus: ***Aus welchen Teilproblemen besteht die gestellte Aufgabe?*** Diese Teilprobleme werden immer weiter aufgegliedert, bis sie "programmierbar" erscheinen.

Wir werden weiter unten in einem Bild noch sehen, dass das Modularisierungsprinzip "Teile und Herrsche" nach dieser Methode arbeitet. Wir können auf dem Bild aber auch erkennen, dass es rekursiv angelegt ist, also durch einfachere Varianten von sich selbst beschrieben wird.

2. Die Bottom-Up-Methode

Beim Vorgehen nach der Bottom-Up-Methode werden zunächst kleine Teillösungen entwickelt, die schrittweise zu größeren Strukturen aufgebaut werden.

Auch hier werden wir in einem späteren Schritt den Zusammenhang dieser Methode mit einem Verfahren, ***das iterativ arbeitet***, erkennen können.

Das Prinzip von "Teile und Herrsche" (divide et impera):

```
procedure LöseDasProblem;
begin
  if Problem_ist_einfach then
    Löse_das_Problem_direkt
  else
  begin
    Zerlege_das_Problem_in_die_Teilprobleme_
    Problem1, Problem2, Problem3 usw.,
    Löse (Problem1); Löse (Problem2); usw.
    Setze die Teillösungen P1, P2, P3 usw. zusammen
  end;
end;
```

Speicherorganisation

Bei rekursiven Programmiertechniken spielt es eine Rolle, in welcher Art und Weise auf Daten zugegriffen werden kann.

Wenn ich den Programmcode (ein kleiner Ausschnitt aus dem Programm HANOI)

```
procedure LiesMirEineListeEin;
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  SL.Clear;
  SL.Add ('Um die Auswahl der Scheiben');
  SL.Add ('jetzt zu beenden,');
  SL.Add ('klicken Sie bitte');
  SL.Add ('den gewünschten Zielplatz');
  SL.Add ('(Platz B oder Platz C)');
  SL.Add ('an.');
  Memo1.Lines := SL;
  SL.Free;
end;
```

vor mir habe, muss ich nicht wissen, wie die Daten intern organisiert sind. Ich kann mich darauf verlassen, dass Delphi aus dem Code, den ich hier eingegeben habe, eine Liste erzeugt, die in ein Memo eingibt und die Liste danach freigibt.

In anderen Fällen - beispielsweise auch bei der Rekursion - ist es dagegen besser, wenn ich mir einige Gedanken darüber mache, was Delphi da im Inneren mit seinen Speichern alles treibt.

Zwei wichtige ***Organisationsprinzipien für Speicherdaten*** muss ich zum besseren Verständnis hier nennen:

1. Die Warteschlange (Queue)

Eine Warteschlange wird immer dann benötigt, wenn von einer Quelle Daten schneller gesendet werden, als sie von einem Empfänger verarbeitet werden können.
Der Tastaturpuffer beispielsweise ist eine Queue. Oft werden Zeichen schneller eingegeben, als der Computer sie verarbeiten kann. Würden die von der Tastatur gesendeten Zeichen nicht in einer Warteschlange eingereiht, so könnten eingegebene Zeichen verloren gehen.

Die Queue arbeitet nach dem Organisationsprinzip **FirstInFirstOut (FIFO)**. Bei einer Warteschlange dürfen deshalb Daten *NUR AM ENDE EINGEFÜGT UND AM ANFANG GELÖSCHT* werden.

2. Der Kellerspeicher (Stack)

Bei der Abarbeitung eines kompilierten Programms enthält der Maschinencode bei Prozeduraufrufen Anweisungen zur Speicherung der Rücksprungadressen auf dem Programmstack. Das hat zur Folge, dass zuerst die innersten Prozeduren oder Funktionen abgearbeitet werden und danach erst die Methoden, die vorher aufgerufen wurden. Zuletzt wird als oberstes Programm das Hauptprogramm wieder aktiviert.

Wenn wir uns diese Arbeitsweise des Kellerspeichers (Programmstack) ansehen, können wir unschwer erkennen, dass der Stack seine Daten in anderer Weise organisiert wie die Queue.

Ein Stack ist ein Datenspeicher, der mit einem Bücherstapel verglichen werden kann. Es kann immer nur mit dem obersten Buch gearbeitet werden. Das Buch, das man zuletzt aufgelegt hat, muss auch als erstes entfernt werden. Will man an das unterste Buch, so müssen die darauf liegenden Bücher, beginnend mit dem obersten Buch, der Reihe nach entfernt werden.

Der Kellerspeicher arbeitet nach dem Organisationsprinzip **LastInFirstOut (LIFO)**. Bei einem Kellerspeicher können deshalb Daten *NUR AM ANFANG EINGEFÜGT UND AM ANFANG AUCH WIEDER GELÖSCHT* werden.

Übrigens: Wollte man auf einer Queue oder auf einem Stack nur einzelne Zeichen verwalten, könnte man das auch mit Hilfe eines Strings oder einer Liste realisieren. Ist die Zugriffsordnung festgelegt, bedeutet das keinerlei Probleme.

Ein Würfelspiel

Pause ist jetzt, jetzt ist Pause ... Jetzt kannst Du eine Zigarette rauchen, einen Kaffee trinken und und und ...
Halt, noch nicht - Zigarette (wenn Du überhaupt willst) und Kaffee kommen gleich. Aber zuerst wollen wir etwas spielen. Spielen entspannt oft besser als alles andere.

Nimm Dir drei schöne quadratische Merkzettel aus Deinem Zettelkasten (oder etwas ähnliches) und lege sie in einer Reihe vor Dich hin: den einen links, den anderen in die Mitte und den letzten rechts. Jetzt brauchst Du nur noch vier hohle Plastewürfel, die man ineinander stecken kann (ich sagte doch: spielen - und das hängt eben auch mit Kinderspielzeug zusammen) oder vier Pappscheiben, eine größer als die andere, oder - wenn Du gar nichts zur Hand hast, gehe an Dein Vermögen und nimm Dir ein Fünfmarkstück, ein Zweimarkstück, ein Markstück und eine Pfennigmünze - wenn du noch alte D-Mark-Münzen haben solltest. Die neuen Euro-Münzen sind da nicht so gut geeignet. So, jetzt hast Du alle Spielutensilien zusammen.

Die Spielregeln sind ganz einfach: Du stellst - ich denke mal, Du hast hohle Plastewürfel - die vier Würfel mit der hohlen Seite nach unten so auf Deinen linken Merkzettel, dass ein kleiner Turm entsteht. Jetzt besteht die Aufgabe darin, den linken Turm auf den rechten Merkzettel umzusetzen. - - Einfach! Nichts leichter als das! - Schon passiert!

Halt! Du darfst immer nur EINEN (!) Würfel bewegen und auf einen anderen Merkzettel stellen. Dabei darf kein Würfel den anderen zudecken, der obere Würfel muss also immer kleiner sein. Den Merkzettel in der Mitte darfst Du als Zwischenlager verwenden.

In wie viel Zügen hast Du den Turm links auf den Merkzettel rechts umgesetzt - schön übereinander und keiner überdeckt einen anderen Würfel? Wenn Du (immer die Spielregeln beachten!) bei vier Würfeln 15 Züge brauchst, bist Du Weltmeister. Darunter kommst Du garantiert nicht! Solltest Du es doch schaffen, wirst Du mit einer Jahresrente von zwei Million Lichtjahren auf die Venus versetzt.

So, jetzt ist wirklich Pause - Spielpause, und der Kaffee kommt auch. Viel Erfolg beim Spiel.

Die Lösung gibt es natürlich auch - wie bei jedem guten Spiel. Aber erst im nächsten Schritt.

Übrigens: Wenn Du willst, kannst Du Dir ja noch ein, zwei oder drei Spielwürfel besorgen und damit Dein Glück probieren. Bei sieben Würfeln musst Du schon mit 127 Zügen rechnen - oder mehr.

Türme von Hanoi

So, jetzt haben wir gespielt. Nun geht es wieder an die Arbeit.

Rufe Dir doch jetzt mal (aus der Datei tutrkalg.exe) die Datei Loesung.exe auf. Zu dieser Datei gibt es keinen Quelltext, Du sollst Dir einfach einmal Bestätigung für Deine gut gewählten und überlegten Züge im Spiel holen.

So, jetzt hast Du gesehen, wie gut Du warst. Und nun kannst Du Dir überlegen, wie Du den Algorithmus für so einen Spielablauf programmieren willst.

Im Schritt Modularisierung haben wir uns überlegt, wie wir an eine größere Programmieraufgabe herangehen könnten. Lies Dir diesen Schritt am besten noch einmal durch.

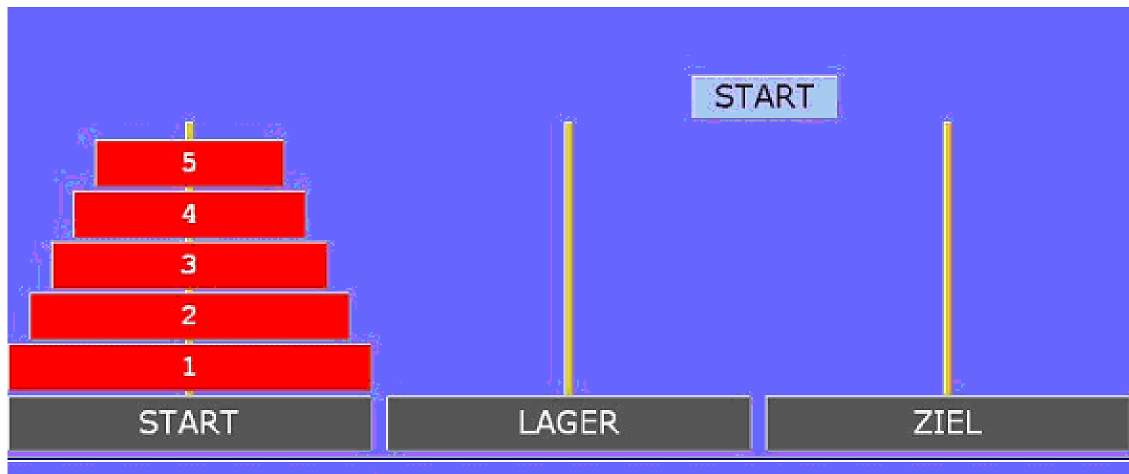
Die Bottom-Up-Methode scheint uns nicht viel zu bringen. Wir sehen erst einmal das Riesenproblem vor uns und können nicht ohne weiteres von kleinen Teillösungen auf größere Strukturen kommen.

Also wollen wir es mit der Top-Down-Methode versuchen.
Hier finden wir auch schon einen Ansatz zu unserem Vorhaben, Probleme rekursiv zu lösen. Das Bild "Teile und Herrsche" (siehe weiter vorn) macht es uns optisch deutlich.

Aber jetzt wird es endlich erst einmal Zeit, die Frage zu stellen: ***Was ist das eigentlich mit dem rekursiven Denken? Was ist das mit den rekursiven Algorithmen??***

Lies Dir jetzt am besten erst einmal die Information in der Datei Legende.hlp (in den Beispieldateien) durch. Durch Anklicken rufst Du sie auf.

Die "Türme von Hanoi" sind ein ausgezeichnetes **Beispiel** für das, was man rekursives Denken nennt. Frei vom Druck einer Weltuntergangsstimmung können wir es uns leisten, mit ganz geringen Scheibenzahlen zu beginnen.



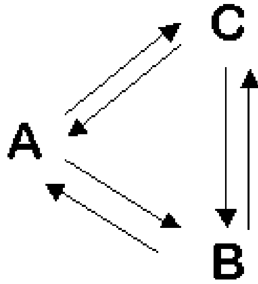
Um keine Scheibe von Nadel START nach Nadel ZIEL zu bringen, ist natürlich keine Umlegung nötig, für eine Scheibe offenbar genau eine Umlegung. Bei zwei Scheiben deponierst Du zuerst die kleine Scheibe auf der Nadel LAGER, bringst dann die größere Scheibe auf Nadel ZIEL und kannst mit dem dritten Zug den Turmbau auf ZIEL mit dem Hinüberlegen der kleineren Scheibe abschließen.

Nehmen wir an, Du weißt, wie n Scheiben umzulegen sind. Wie geht es dann mit n+1 Scheiben weiter? Genau, wie eben beschrieben. Du baust den Turm aus den n Scheiben im Zwischenlager (LAGER) statt im ZIEL auf, was ich als bekannt unterstelle, um dann die unterste (n+1)-Scheibe auf der Nadel ZIEL abzulegen. Jetzt kannst Du mühelos den zwischengelagerten Turm aus n Scheiben auf die größte Scheibe versetzen. - Damit ist das Grundprinzip klar.

Ratsam ist es, jetzt noch einmal zum Spiel mit den Würfeln zurückzukehren. Lass Dir das eben Gesagte langsam "auf der Zunge zergehen" und spiele es mit den Würfeln nach (aber nicht auf der Zunge).

Eine praktikable Regel ist folgende:

Die drei Stangen (Nadeln) werden in Dreiecksform angeordnet. Es müssen nicht unbedingt "Diamantnadeln" sein (wie in der Legende).



Du legst die kleinste Scheibe von Nadel A auf Nadel B (gerade Zahl von Scheiben) oder auf Nadel C (ungerade Zahl von Scheiben) und bewegst diese Scheibe nur in dem hierdurch festgelegten Umlaufsinn (ABCAB ... bzw. ACBAC ...) Die nächste Scheibe legst Du so, dass die kleinste Scheibe nicht berührt wird, und behältst den Umlaufsinn bei. Wenn Du die Scheiben von oben nach unten durchnummerierst (anders wie im obigen Bild), so wandern die Scheiben mit ungerader Nummer wie die kleinste Scheibe über die Nadeln, während sich die restlichen Scheiben entgegengesetzt bewegen.

Wie viele Umlegungen werden für die Scheiben benötigt?
Um n Scheiben von A nach C zu bringen, sind 2 hoch n - 1 Umlegungen nötig, für acht Scheiben sind das immerhin schon 255 Umlegungen und für 64 Scheiben (die Zahl in der Legende) ist das eine nicht vorstellbare 20-stellige Zahl. Da haben die Priester im Tempel von Benares aber noch zu tun!

Um es genauer zu sagen: Für 64 Scheiben wären also 2 hoch 64 - 1 Umsetzungen nötig. Dies sind ungefähr 18,45 Trillionen (18,45*10 hoch 18) Umsetzungen. Wenn man für eine Umsetzung 20 Sekunden rechnet, so würden bis zur Lösung der Aufgabe etwa 11,7 Billionen Jahre vergehen.

Übrigens: Wer sich im Rechnen üben will, kann sich ja solche Fragen wie "Wie sieht die Verteilung von n Scheiben nach dem p-ten Zug aus?" oder "Wie oft und wann wird beim Umsetzen eines Turmes von n Scheiben die k-te Scheibe bewegt?" vornehmen. Wenn Du Freude an solchen Fragen hast, dann viel Spaß. Wenn nicht, geht die Welt auch nicht unter (die Priester sind noch nicht so weit).

Rekursive Prozeduren

Im vorigen Schritt hieß es so gegen Ende:

Du baust den Turm aus den n Scheiben im Zwischenlager (LAGER) statt im ZIEL auf, was ich als bekannt unterstelle, um dann die unterste (n+1)-Scheibe auf der Nadel ZIEL abzulegen. Jetzt kannst Du mühelos den zwischengelagerten Turm aus n Scheiben auf die größte Scheibe versetzen. - Damit ist das Grundprinzip klar.

Bei diesen Überlegungen haben wir uns auf die Bottom-Up-Methode gestützt: kleine Teillösungen => große Strukturen; von n zu n+1. Aber am Anfang von Schritt 4 waren wir uns einig darüber, dass wir es "mit der Top-Down-Methode versuchen" wollten; also Riesenproblem in kleine Teillösungen zerlegen, bis diese "programmierbar" erscheinen; also von n zu n-1.

Wir fassen diese Überlegungen in einem Algorithmus zusammen:

Arbeitsauftrag: Bewege n Scheiben von der Nadel START auf die Nadel ZIEL.

1. **Wenn n > 1**, erteile folgenden Arbeitsauftrag: // Abbruchkriterium
   Bewege n-1 Scheiben von der Nadel START auf die Nadel LAGER.
2. Bringe eine Scheibe von der Nadel START auf die Nadel ZIEL.
3. **Wenn n > 1**, erteile Arbeitsauftrag:
   Bewege n-1 Scheiben von der Nadel LAGER auf die Nadel ZIEL.

Bitte beachte, dass für jeden Arbeitsauftrag START, ZIEL und LAGER die Plätze wechseln.

Um diese Arbeitsaufträge realisieren zu können, verwenden wir folgende Prozedur, die diese Überlegungen widerspiegelt:

```
procedure bewege (n: Integer; Start, Ziel, Lager: Char);
begin
 if n > 1 {unsere Abbruchsbedingung} then
   bewege (n-1, Start, Lager, Ziel);      {Rekursion => Selbstaufruf}

 { hier kommt die Ausgabe "bewege [Start] nach [Ziel]" }

 if n > 1 {unsere Abbruchsbedingung} then
   bewege (n-1, Lager, Ziel, Start)       {Rekursion => Selbstaufruf}
end; {der Prozedur bewege}
```

Um die Prozedur ausprobieren zu können, rufe das Programm Motor.exe (aus der Datei TutExe.exe) auf. Dabei wird folgende Ausschrift angezeigt:

```
1  Bewege den Würfel von <A> nach <B>.
2  Bewege den Würfel von <A> nach <C>.
3  Bewege den Würfel von <B> nach <C>.
4  Bewege den Würfel von <A> nach <B>.
5  Bewege den Würfel von <C> nach <A>.
6  Bewege den Würfel von <C> nach <B>.
7  Bewege den Würfel von <A> nach <B>.
```

Tatsächlich liefert die einfache kurze Prozedur das gewünschte Ergebnis.

Das Beispiel zeigt, wie kurz und leistungsfähig rekursive Prozeduren sein können.

Übrigens: Den Quelltext für die Prozedur bewege findest Du in uMotor.pas.

Rekursion

Im tiefen Keller sitz' ich hier und - kann mich vor Daten nicht mehr retten. Sieh Dir deshalb noch einmal Schritt Speicherorganisation an.

1. Das allgemeine Prinzip der rekursiven Problemlösung

Zur Lösung des Problems "Türme von Hanoi" haben wir einen rekursiven Algorithmus formuliert, der stets einfachere Varianten von sich selbst enthält. Die Bearbeitung des Falles n = 5 wurde mit der einfacheren Variante n - 1 = 4 beschrieben usw. Das ist das ***Prinzip des rekursiven Algorithmus***.

Ein Algorithmus ist ein rekursiver Algorithmus, wenn er mit Hilfe einfacher Varianten von sich selbst beschrieben wird.

Die Prozedur bewege aus Schritt Rekursive Prozeduren ist die direkte programmtechnische Umsetzung des zunächst bildhaft formulierten rekursiven Algorithmus für das Stapeln von Scheiben. Da der Algorithmus rekursiv ist, ist auch die direkte Übersetzung in die Prozedur bewege rekursiv.

Man erkennt die Rekursion daran, dass sich die Prozedur bewege selbst aufruft.

Wir nennen Prozeduren, die sich selbst aufrufen, rekursive Prozeduren bzw. rekursive Funktionen.

In unserem Beispiel der "Türme von Hanoi" ist die einfachste Variante des Algorithmus für das Stapeln von n Scheiben der Fall n = 1. Bei n = 1 dürfen keine weiteren rekursiven Aufrufe von bewege erfolgen.

In unserer Prozedur erfolgt deshalb nur dann ein weiterer Aufruf innerhalb der Prozedur, wenn n größer als 1 ist. Hier ist eine ***Abbruchbedingung*** für die rekursive Prozedur formuliert, ähnlich der Abbruchbedingungen in einer Schleife. Natürlich muss sichergestellt werden, dass die Abbruchbedingung auch erreicht wird. Andernfalls käme es zu einer ***unendlichen Rekursion***!!

Jeder rekursive Algorithmus und jede rekursive Prozedur muss eine (wirklich zu erreichende) Abbruchbedingung enthalten.

Um die Wirkungsweise der Rekursion besser verstehen zu können, einigen wir uns auf zwei Begriffe:

1. Die Bestimmung der nächsteinfacheren Variante eines Problems nennen wir Rekursionsschritt.
2. Die nach einem Rekursionsschritt erreichte Variante nennen wir Rekursionsstufe.

Probleme, die ihrer Natur nach zur Strategie "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 zuletzt 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.

2. Techniken der rekursiven Programmierung

Der in Schritt Speicherorganisation genannte Kellerspeicher (Stack) ist ein Speicherbereich, der auf besondere Art und Weise organisiert ist: gleich einem Bücherstapel kann ein Speicherwert *NUR OBEN AUFGELEGT UND AUCH NUR OBEN WIEDER ABGENOMMEN* werden.

Um besser verstehen zu können, wie Rekursion programmtechnisch funktioniert, sehen wir uns die nachfolgende Skizze an.

Die Skizze zeigt die ***Prozedur LiesZeichen*** mit einer lokalen Variablen Z. Diese Prozedur liest bei ihrem Aufruf EIN Zeichen ein. Das Abbruchskriterium ist das Zeichen **|#|**.

Auf dieser Skizze sind von der ersten Rekursionsstufe aus drei Rekursionsschritte dargestellt; das bedeutet, die Prozedur hat sich nach ihrem ersten Aufruf noch dreimal selbst aufgerufen. Die Abbruchsbedingung wurde auf Rekursionsstufe (4) eingegeben. Die Rekursionsstufen (1) bis (4) sind voneinander abgesetzt dargestellt.

Auf der letzten Zeile wird die Arbeitsweise der Prozedur durch die Bildschirmausschrift belegt.

```
========================================================================
                                                +-----------------------
(1) LiesZeichen                                 ||#| = Abbruchbedingung
Eingabe: Z <= |R|                               +-----------------------
            +-----------------------------------------------------------
Z <> |#| =====> (2) LiesZeichen
            |   Eingabe: Z <= |O|
            |               +-------------------------------------------
            |   Z <> |#| =====> (3) LiesZeichen
            |               |   Eingabe: Z <= |M|
            |               |               +---------------------------
            |               |   Z <> |#| =====> (4) LiesZeichen
            |               |               |   Eingabe: Z <= |#|
            |               |               |   Z = |#| ABBRUCH !
            |               |               |   (4) Ausgabe |#|
            |               |               +---------------------------
            |               |   (3) Ausgabe |M|
            |               +-------------------------------------------
            |   (2) Ausgabe |O|
            +-----------------------------------------------------------
(1) Ausgabe |R|

========================================================================

Bildschirmausgabe:  R O M # # M O R

========================================================================
```

Um die Prozedur LiesZeichen auch ausprobieren zu können, liegt die Turbo-Pascal-Datei rekursio.pas den Beispieldateien bei. Sie zeigt nur die unbedingt zum Verständnis notwendigen Daten. Die Datei Rekursio.exe (in TutExe.exe), die nur im DOS-Bildschirm läuft, kann durch Anklicken aufgerufen werden. Hier kannst Du selber probieren.

Und nun steht die große Frage im Raum:

Wie werden die einzelnen Zeichen gespeichert, wenn wie hier nur EINE Variable in der Prozedur vorhanden ist?

Die Frage lässt sich insofern leicht beantworten, wenn Du Dir klar machst, dass die Variable Z eine ***lokale Variable*** innerhalb der Prozedur LiesZeichen ist.

Für lokale Variablen werden erst bei Prozeduraufruf Speicherzellen auf dem Stack zur Verfügung gestellt und erst dann wieder freigegeben, wenn die Abarbeitung der Prozedur beendet ist.

Das hat zur Folge, dass bei jedem Prozeduraufruf für die Variable Z auf dem Stack andere Speicherzellen verwendet werden - die allerdings auf jeder Rekursionsstufe den gleichen Namen haben.

Wenn Du mehr über lokale Variablen und ihren Geltungsbereich wissen möchtest, kann ich Dir nur "Pascal-Grundlagen * Variablen und Konstanten" hier auf DSDT empfehlen.

Auch in der Prozedur bewege aus Schritt Rekursive Prozeduren funktioniert die rekursive Programmierung nur, weil die Werteparameter der Prozedur bewege wie lokale Variablen behandelt werden.

Und hier die Prozedur zur Erinnerung in Kurzform:

```
procedure bewege (n: Integer; Anfang, Ende, Mitte: Char);
begin
  if n > 1 then bewege (n-1, Anfang, Mitte, Ende);
  if n > 1 then bewege (n-1, Mitte, Ende, Anfang);
end {bewege};
```

Du siehst, in der Kürze liegt die Würze. Und die Hauptarbeit leisten die Werteparameter. Beachte dabei, dass die Reihenfolge der Parameter stets geändert ist. Darin liegt das Geheimnis dieser Prozedur!!

Das muss ich hier doch noch einfügen:

Ich bin immer wieder erstaunt darüber, wie kurz rekursive Prozeduren oder Funktionen sein können. Eigentlich wird die Kopfleiste der Prozedur mit Parametern dreimal untereinander geschrieben - freilich mit veränderter Parameter-Reihenfolge - und dann hat es sich schon. Und wo ist der eigentliche Quellcode? Du verstehst sicher, dass mich das in Erstaunen versetzt, zumal das Ergebnis so ungeheuer ist. Da würde ich auf Befragen einen Quelltext von mehreren hundert Zeilen erwarten.

Rekursive Programmierung ist doch etwas Tolles. Auch wenn sie so "harte Nüsse" bereitstellt. Also: (siehe Einführung) Geduld lohnt sich!!

Aber jetzt noch einmal:

Werteparameter von Prozeduren und Funktionen werden bei der rekursiven Programmierung wie lokale Variablen verwaltet.

Übrigens: Auch über Werteparameter und Variablenparameter kannst Du Dich wenn nötig bei DSDT informieren: Pascal-Grundlagen * Prozeduren und Funktionen.

Iteration

Neben der Rekursion steht die Iteration als gleichwertiges Verfahren zur Problemlösung bereit. Was es damit auf sich hat, soll dieser Schritt klären.

Jetzt wollen wir uns nochmals an den Anfang erinnern. Dort war von der Bottom-Up-Methode die Rede gewesen. In diesem Zusammenhang habe ich behauptet, dass wir "in einem späteren Schritt den Zusammenhang dieser Methode mit einem Verfahren, das iterativ arbeitet, erkennen können."
Gleich muss ich den Beweis dazu antreten.

Wenn die rekursive Problemlösung doch etwas so Tolles ist, wie ich eben behauptet habe, hat sie aber bei alledem auch Nachteile. Nicht jeder rekursive Algorithmus muss auch rekursiv programmiert werden. Das alternative Verfahren zur Rekursion ist die Iteration, die Berechnung durch wiederholtes Anwenden einer Operation in einer Schleife. Und da sind wir wieder der Bottom-Up-Methode sehr nahe.

Zunächst eine kleine Spielerei mit Buchstaben und Wörtern.

Wie viel verschiedene Wörter kann man aus den vier Buchstaben **A, M, O, R** bilden?

Lösung:

AMOR	AMRO	AOMR	AORM	ARMO	AROM
MAOR	MARO	MOAR	MORA	MRAO	MROA
OAMR	OARM	OMAR	OMRA	ORAM	ORMA
RAMO	RAOM	RMAO	RMOA	ROAM	ROMA

Zuerst gibt es vier Möglichkeiten, den ersten Buchstaben zu wählen. Ist dieser gewählt, so gibt es nur noch drei Möglichkeiten, den zweiten Buchstaben zu wählen. Für den dritten Buchstaben sind noch zwei Möglichkeiten vorhanden und nur noch eine für den letzten Buchstaben.

Da die Wahlmöglichkeiten unabhängig voneinander sind, gibt es insgesamt

4 * 3 * 2 * 1 = 24 Möglichkeiten.

In der Mathematik - Du weißt es - nennt man das Produkt 4 * 3 * 2 * 1 auch "Fakultät von 4" und schreibt dafür 4! (vier Fakultät).

So ist beispielsweise:

1! = 1   3! = 3 * 2 * 1   5! = 5 * 4 * 3 * 2 * 1

2! = 2 * 1   4! = 4 * 3 * 2 * 1   n! = n * (n-1) * (n-2) * ... * 1

Daraus folgt die allgemeine rekursive Definition für n! :

Definition für n-Fakultät:

Für n > 1 gilt n! = n * (n-1)!

Für n = 1 gilt n! = 1

Du siehst:

n! wurde mit einer einfacheren Variante von sich selbst, nämlich mit (n-1)! definiert. Also 5! durch 5*4! und dann 4! durch 4 * 3! usw. bis wir bei 1! = 1 angekommen sind. Dann muss die Rekursion wegen der Bedingung n > 1 abbrechen.

Wir setzen diese Definition in die folgende Funktion um:

```
function fakrek (n: Integer): LongInt;
begin
```

```
  if n > 1 {die Abbruchbedingung} then
    Result := n * fakrek (n-1)
  else
    Result := 1;
end; {der Funktion fakrek}
```

Da der Funktionswert schnell sehr groß werden kann, verwenden wir für das Funktionsergebnis den Wert LongInt (-2147483648..2147483647 / 32 Bit einschließlich Vorzeichen). Trotzdem liefert die Funktion für n > 12 falsche Ergebnisse, da für n > 12 auch der Wertebereich von LongInt überschritten wird.

Die Funktion kannst Du im Programm fak1.exe (in der Datei TutExe.exe) ausprobieren. Dort gibt es auch den Quelltext (ufak1.pas).

Obwohl die mathematische Definition von n! rekursiv ist - wie wir eben festgestellt haben -, können wir mit Hilfe einer einfachen FOR-Schleife die Rekursion vermeiden.

Die Funktion dazu kommt hier:

```
function fakitera (n: Integer): LongInt;
var I: Integer;
    F: LongInt;
begin
  F := 1;
  for I := 2 to n do F := F * I;
  Result := F;
end; {der Funktion FakItera}
```

Bei der Berechnung von n! gehen wir jetzt, getreu der Bottom-Up-Methode, vom einfachsten zum komplizierteren. Deshalb beginnen wir in der Funktion fakitera mit F := 1. Für alle natürlichen Zahlen i von 2 bis n wird der neue Wert von F jeweils aus dem alten Wert von F durch die Anweisung F := F * I berechnet. Die Anweisung F := F * I wird also (n-1)-mal nacheinander ausgeführt.

Eine kurze Zusammenfassung:

Das in Funktion fakitera angewandte Verfahren ist nicht rekursiv sondern ***iterativ***. Im Gegensatz zur Rekursion beginnt man bei der Iteration mit der einfachsten Stufe und berechnet in einer Schleife aus der vorangegangenen Stufe die nächsthöhere, solange bis das Ziel erreicht ist.

Natürlich muss auch jeder iterative Algorithmus und jede iterative Prozedur oder Funktion eine ***Abbruchbedingung*** haben, die auch erreicht wird.
Bei der Iteration wird der Rekursionsschritt umgekehrt. Während man bei der Rekursion von einer Stufe zu einer einfacheren Stufe zurückgeht (Rekursionsschritt), geht man bei der Iteration von einer Stufe zur nächsthöheren (Iterationsschritt), um dann die nächstkompliziertere Iterationsstufe zu erreichen.
Bei der rekursiven Berechnung von n! realisiert die Anweisung

```
Fak := N * Fak(n-1)
```

den Rekursionsschritt.

Bei der iterativen Berechnung von n! realisiert die Anweisung

Fak := Fak * n

den Iterationsschritt.

Unter ufak1.pas und ufak2.pas kannst Du beide Programmierverfahren nochmals miteinander vergleichen.

Der italienische Mathematiker ***Leonardo Fibonacci*** (um 1170 bis ca. 1240), der das mathematische Wissen des klassischen europäischen, arabischen und indischen Kulturkreises zusammentrug und durch Beiträge zur Algebra und Zahlentheorie ergänzte, hat eine Zahlenreihe in der folgenden Form, die sogenannte ***Fibonacci-Reihe***, entwickelt, die in der Mathematik weit verbreitet ist:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55 usw.

Wer gern knobelt, kann ja in der Zwischenzeit überlegen, welchem mathematischen Gesetz diese Zahlenreihe folgt. Wenn Du keine Lust dazu hast - auch nicht schlimm. Du erfährst es sowieso gleich.

Das Geheimnis dieser Zahlenreihe beruht auf der Tatsache, dass eine Zahl in dieser Reihe stets die Addition der beiden vorhergehenden Zahlen ist; dabei sind die erste und die zweite Zahl der Reihe jeweils mit dem Wert 1 belegt.

Anhand dieser Zahlenreihe könnten wir Rekursion und Iteration auf interessante Weise miteinander vergleichen und die Unterschiede zwischen beiden Programmiertechniken deutlich herausstellen. Aber da das auch wieder nur eine Sache für Interessierte ist und Du sicher nochmals eine Verschnaufpause brauchst, will ich für alle, die Interesse an diesem Vergleich haben, noch ein Programm zur Verfügung stellen, das uns den Unterschied in der Verarbeitungsdauer verdeutlichen kann (Beispieldatei: uFibona.pas).

In der Informatik gilt ein bewiesener Satz, so habe ich gelesen, dass es zu einer rekursiven Problemlösung immer auch eine iterative gibt.

Theoretisch könnten wir nach diesem Satz auf die rekursive Lösung ganz verzichten. Doch das ist wirklich nur Theorie. Man muss nämlich erst einmal für eine bestimmte Klasse von Problemen einen iterativen Algorithmus finden. So ist zum Beispiel für das Problem der "Türme von Hanoi" eine iterative Lösung grundsätzlich möglich, jedoch außerordentlich schwierig. Mir ist nicht bekannt, dass es so eine Lösung gibt.

Wegen der nur angerissenen möglichen Nachteile der Rekursion sollte sie immer dann vermieden werden, wenn eine iterative Lösung offensichtlich ist. Lässt sich dagegen ein rekursiver Algorithmus leichter finden und elegant formulieren und spielt die Rechenzeit keine Rolle, so ist die Rekursion vorzuziehen.

Ein interessantes Beispiel für Rekursion wäre auch ein Sortieralgorithmus nach dem Grundsatz "Sortieren durch Mischen (MergeSort)".
Nach dem Prinzip von "Teile und Herrsche" erhält man einfachere Varianten eines zu sortierenden Feldes, indem man es in ungefähr gleiche Teile aufteilt, jede Hälfte getrennt sortiert und dann die Hälften zusammenmischt.
Mit diesem Ansatz lässt sich ein Rekursionsschritt wie folgt skizzieren:

- Sortiere die erste Hälfte des Feldes
- Sortiere die zweite Hälfte des Feldes
- Mische beide Hälften zusammen.

Die Programmierung dieses Beispiels überlasse ich Dir, wenn Du das willst. Die Richtung habe ich ja angegeben.

Übrigens: Es gibt Probleme, für die sich ein Lösungsalgorithmus nicht direkt angeben lässt. Sofern es überhaupt eine Lösung gibt, kann man sie höchstens durch systematisches Ausprobieren finden. In der Informatik spricht man von ***Backtracking-Verfahren***.

Auch bei diesem Programmierverfahren, das einen Zugang zu einer ganz neuen Klasse von Problemlösungsmethoden liefert, spielt die Rekursion eine wichtige Rolle.
Darüber vielleicht einmal später.

Zusammenfassung

Wir haben die Rekursive Programmiertechnik als ein Werkzeug kennen gelernt, mit dem sehr komplexe Aufgabenstellungen durch erstaunlich kurze und einfache Algorithmen programmtechnisch sauber gelöst werden können.

Die in Schritt Rekursive Prozeduren erarbeitete Prozedur ***bewege*** - im beiliegenden Beispielprogramm Hanoi nennt sie sich ***transportiere*** - stellte uns einen Rekursionsalgorithmus und seine Wirkungsweise in seiner Funktion vor.

Rekursives Denken lässt sich nur schwer "verstehen", da es eine völlig ungewohnte Denkweise ist.
Nach dem Grundsatz "learning by doing" habe ich diesem Tutorial ein Anschauungsprogramm "Die Türme von Hanoi" (Hanoi.exe bzw. als Projektdatei Hanoi.dpr) beigefügt. Das gleichnamige Projekt in delphi-source.de hat mir den letzten Anstoß gegeben, das schon lange in Angriff genommene Projekt in eine endgültige Form zu bringen.
Grundsatz für dieses Vorhaben war, ein recht übersichtliches, klar strukturiertes Programm zu entwickeln. Auch sollten die im Programm verwendeten Daten und andere Angaben möglichst redundanzfrei gebraucht werden; das heißt, wenn Daten im Zuge einer Erweiterung des bisherigen Programms zu ändern sind, dann soll dies NUR AN EINER STELLE notwendig sein. Das allein schließt schon manche Fehlerquellen aus. Außerdem wollte ich durch besondere Maßnahmen mögliche Fehler weitgehendst vermeiden. Und nicht zuletzt wollte ich selbst dem Phänomen Rekursion etwas näher kommen; denn auch für mich gilt: "learning by doing".

Das Programm Hanoi kannst Du gern erweitern, wie ich es im Sourcecode schon angeboten habe: den manuellen, also von Dir bestimmten Programmablauf habe ich bereits vorgesehen. Alle dafür notwendigen Elemente sind vorhanden.
Sollte Dir dieses Programm Anregungen und Hilfen zum eigenen Programmieren geben, würde ich mich freuen. Solltest Du aber meinen: hier kann ich etwas anders und besser machen, so lass es mich bitte wissen. Meine eMail-Adresse findest Du unten.

Den Sinn eines solchen Tutorials bzw. eines ins Netz gestellten Programms sehe ich auch darin, über manche Programmiererfahrungen zu sprechen (wozu haben wir denn sonst unsere eMails) und so die Erfahrung zu machen: hier ist eine oder einer, die haben sich auch gemüht, dem Laufzeitfehlerteufel ein Schnippchen zu schlagen. Und das ist doch manchmal ganz hilfreich. Und unserem Forum tut es sicher auch gut.