Rekursion

in Pascal, C und BASIC

Karel Štípek

Stellen Sie sich vor, Sie sollten folgende Aufgabe lösen:

```
  SEND
+ MORE
------
 MONEY
```

Die Buchstaben sollen durch Ziffern so ersetzt werden, daß das Beispiel mathematisch stimmt.

Als ich diese Aufgabe bekommen habe, hatte ich Lust ein allgemeines Programm zu schreiben, mit dem man auch andere ähnliche Aufgaben lösen kann. Prinzipiell geht es darum, alle möglichen Ziffernkombinationen zu testen und die richtige Lösung anzuzeigen. Gut, nehmen wir Turbo-Pascal zu Hilfe und beginnen wir. Anfang ist nicht schwer:

PASCAL

```
program algebgr; uses dos, crt;
```

In unserem Beispiel gibt es 8 verschiedene Buchstaben, diese Zahl definieren wir als Konstante MAXB. Die 8 möglichen Ziffern fassen wir zu einem Array B zusammen. Alle Buchstaben können Werte von 0 bis 9 haben, nur S und M (Anfang des Wortes) sollten größer Null sein. Die Grenzwerte werden in den Arrays VON und BIS definiert. Jede Ziffer darf in der Kombination nur einmal auftreten, welche verwendet wird, speichern wir im Array USED.

```
const maxb = 8;                    { S E N D M O R Y }
      von:  array[1..maxb] of byte = (1,0,0,0,1,0,0,0);
      bis:  array[1..maxb] of byte = (9,9,9,9,9,9,9,9);
      used: array[0..9] of boolean = (false, false, false,
            false,false,false,false,false,false,false);
      var b:   array[1..maxb] of longint;
```

Kern des Programms stellt die Funktion TEST dar. Die wird alle möglichen Ziffernkombinationen erstellen und testen. Parameter I bezeichnet das Index im Array B, also Reihenfolge des Buchstabens. J und K sind lokale Hilfsvariablen.

```
procedure test(i:integer);
var j,k: byte;
begin
```

Der Wert des gerade bearbeiteten Buchstabens wird sich im vordefinierten Intervall ändern.

```
  for j:= von[i] to bis[i] do
```

Der Wert ist nur dann zulässig, wenn die Ziffer in der Kombination noch nicht auftritt, in dem Fall wird er im Arrayelement B gespeichert und die Ziffer wird in USED „besetzt“.

```
    if not used[j] then begin
      b[i]:= j; used[j]:= true;
```

Jetzt kommt der wichtigste Trick, der uns viel Code erspart. Das gleiche, was wir mit dem ersten Buchstaben gemacht haben, machen wir auch mit dem nächsten, solange wir nicht beim letzten sind. Die Prozedur TEST ruft sich selbst neu auf - eine sogenannte Rekursion.

```
      if i < maxb then TEST(i+1)
```

Wenn wir den letzten Buchstaben bearbeiten, zeigen wir einfach die erstellte Kombination an (immer in der gleichen Zeile)

```
        else begin write(#13);
        for k:= 1 to maxb do write(b[k]:2);
```

und testen, ob die aktuellen Buchstabenwerte unsere Bedingung erfüllen.

```
        if ((b[1]*10 + b[2])*10 + b[3])*10 + b[4] +
           ((b[5]*10 + b[6])*10 + b[7])*10 + b[2] =
           (((b[5]*10 + b[6])*10 + b[3])*10 + b[2])*10 + b[8]
then
```

Wenn ja, dann schreiben wir neben der Ziffern „OK“ und setzen auf der nächsten Zeile fort.

```
             begin write('  OK'); writeln; end;
        end;
```

Nachdem eine Ziffer in allen Kombinationen mit den weiter rechts liegenden Ziffern verwendet wurde, muß sie wieder „freigesetzt“ werden.

```
      used[b[i]]:= false; end;
end; {test}
```

Das Hauptprogramm besteht aus einer einzigen Zeile - dem Aufruf der Funktion TEST ab dem ersten Buchstaben.

```
begin
  test(1);
end.
```

Auf dem Bildschirm sehen Sie eine Zeile von Ziffern, die sich schnell ändern. Immer, wenn eine gültige Lösung gefunden wird, erscheint die Kombination mit „OK“ und die Anzeige wird auf der nächsten Zeile fortgesetzt.

C

Bei der Übersetzung dieses Programms in C gibt es keine nennenswerten Besonderheiten.

```
/* REKURS.CPP */
/* VISUAL-C++, Vers. 4.1 */

#include <stdio.h>

#define TRUE -1
#define FALSE 0
const int MAXB = 9;
char von[] = {0, 1, 0, 0, 0, 1, 0, 0, 0, 0}; // S E N D M O R Y
char bis[] = {9, 9, 9, 9, 9, 9, 9, 9, 9, 9};
int used[10] = {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE };
long b[MAXB];

void test(int i)
{
  char j, k;
  for (j=von[i]; j<=bis[i]; j++)
  {
    if (!used[j])
    {
      b[i] = j;
      used[j] = TRUE;
      if (i<MAXB)
      {
        test (i + 1);  // Rekursion
      }
      else
      {
        printf("\x0d");
        for (k=1; k<=MAXB; k++)
        {
          printf("%i ",b[k]);
        }
        if (((b[1] * 10 + b[2]) * 10 + b[3]) * 10 + b[4] + // SEND
            ((b[5] * 10 + b[6]) * 10 + b[7]) * 10 + b[2] == // MORE
            (((b[5] * 10 + b[6]) * 10 + b[3]) * 10 + b[2]) * 10 + b[8]) // MONEY
        {
          printf(" OK\n");
        }
      }
      used[b[i]] = FALSE;
    }
  }
}

void main(void)
{
  test(1);
}
```

BASIC

Die Übersetzung des Programms in BASIC war bezüglich der Initialisierung der Arrays etwas abweichend. Uninitialisierte Arrays wie etwa `used` oder `b` werden mit dem richtigen Typ deklariert. Bei den Arrays, die einen Anfangswert erfordern (`von, bis`) müßte man die Anfangswerte im Rahmen einer `For-Next`-Schleife setzen, was nicht besonders elegant ausschaut. In diesem Beispiel wurden daher die Variablen `von` und `bis` als Typ Variant gewählt, der die Initialisierung mit der Funktion `Array` erlaubt. Für optimale Laufgeschwingigkeit sollte man aber den Typ Variant vermeiden.

Als Testausgabe wurde das Objekt `Debug` mit der Methode `Print` verwendet. Zur Beschleunigung des Ablaufs wurde aber die laufende Ausgabe jedes Versuchs ausgeklammert. Es werden nur die gültigen Lösungen ausgegeben.

```
'VISUAL BASIC for APPLICATIONS (ACCESS)
Option Explicit
Const maxb As Integer = 9                    ' S E N D M O R Y
Dim von As Variant
Dim bis As Variant
Dim used(0 To 9) As Boolean
Dim b(1 To maxb) As Long
Dim i As Byte

Public Sub test(i As Integer)
  Dim j, k As Byte
  For j = von(i) To bis(i)
    If Not used(j) Then
      b(i) = j
      used(j) = True
      If i < maxb Then test (i + 1)
      Else
        'For k = 1 To maxb
        '  Debug.Print (b(k));
        'Next k
        'Debug.Print
        If ((b(1) * 10 + b(2)) * 10 + b(3)) * 10 + b(4) + _
           ((b(5) * 10 + b(6)) * 10 + b(7)) * 10 + b(2) = _
           (((b(5) * 10 + b(6)) * 10 + b(3)) * 10 + b(2)) * 10 + b(8) Then
          For k = 1 To maxb
            Debug.Print (b(k));
          Next k
          Debug.Print "  OK"
        End If
      End If
      used(b(i)) = False
  Next j
End Sub

Public Sub testtest()
  von = Array(0, 1, 0, 0, 0, 1, 0, 0, 0, 0)
  bis = Array(9, 9, 9, 9, 9, 9, 9, 9, 9, 9)
  test (1)
End Sub
```

□

CDROM „Delphi Programmiertools“

International Thomson Publishing, ISBN 3-8266-0230-7, öS 233,-

Hans Lohninger

Das CDROM „Delphi Programmiertools“ enthält eine bunte Mischung aus Free- und Shareware und einigen wenigen Demos („Crippleware“), insgesamt 80 Produkte. Die enthaltene Software kann man grob in zwei Kategorien einteilen: zum einen Delphi-Komponenten und -Erweiterungen, und zum anderen allgemeine Zubehör-Software für Programmierer.

Mit den 80 Produkten (die übrigens das CDROM nur zu 8 % belegen, der Rest ist leeres Plastik) wird ein 'CD-Manager' mitgeliefert, der helfen soll, einen besseren Überblick zu den angebotenen Produkten zu bekommen. Man kann sich mit Hilfe des CD-Managers die jeweiligen (deutschen) Hilfetexte ansehen und bei Interesse das jeweilige Software-Produkt auf die Harddisk kopieren.

Dieser CD-Manager ist im Prinzip eine gute Sache. Allerdings ist er wohl etwas zu mager geraten, sodaß der genervte Benutzer sich schnell ein kleines Batch-Programm bastelt, das alle Hilfetexte in ein File kopiert - womit der Überblick wieder da ist. Störend ist auch, daß der CD-Manager anscheinend nicht auf Systemen mit veschiedenen Grafik-Auflösungen getestet worden ist: Bei einer Auflösung von 1024x768 passen einige Elemente der Oberfläche des CD-Managers nicht mehr zusammen.

Die Auswahl der auf der CD enthaltenen Produkte erfolgte meiner Ansicht nach eher nachlässig. So gibt es insgesamt zwar 4 verschiedene Texteditoren im Angebot, allerdings ist zum Beispiel der sehr gute PFE nur in der 32-Bit Version enthalten. Verglichen mit dem Platz, der noch auf der CD frei ist (immerhin runde 600 MB), hätten die Produzenten durchaus noch das eine oder andere Delphi-Juwel mit auf das CDROM nehmen können.

Der Wert des CDROMs wird auch insofern relativiert, als man sich für dessen stolzen Preis von 233,- öS bei entsprechender Hardware-Ausstattung ca. 60 MB aus dem Internet holen kann - also um circa 10 MB mehr als auf der CD vorhanden ist. □

Wie die verschiedenen Programmierer ihre Fahrräder bauen

ADA-Programmierer bauen ein viereckiges Rad und passen alle Straßen an.

ALGOL-Programmierer weigern sich, Räder zu bauen, weil es sie irgendwohin bringen könnte.

APL-Programmierer schweben in höheren Sphären, sie brauchen keine Räder.

ASSEMBLER-Programmierer bauen tausende von Rädern, keine 2 passen auf eine Achse.

BASIC-Programmierer bauen nur ein Rad, aber finden keine Achse dazu.

COBOL-Programmierer bauen TAUSEND-RÄDRIGE-TRANSPORT-MODULE und verbieten das Gehen.

FUNKTIONAL-Programmierer rufen eine Funktion HOLZ auf und hoffen, ein Rad zu bekommen.

FORTH-Programmierer bauen Räder und vergessen, wo sie sie gestapelt haben.

FORTRAN-Programmierer werden wahnsinnig bei der Suche nach Rädern, die mit "I" beginnen.

LOGO-Programmierer bauen kleine rote Autos.

LISP-Programmierer (bauen Räder mit (Rädern mit/Rädern mit (Rädern mit (dem was LISP-Programmierer bauen))))).

MICRO-Programmierer wissen nicht, daß Räder existieren.

PASCAL-Programmierer erklären das Gehen zur Tugend.

PL/I-Programmierer setzen ein Team ein, um eine Räderfabrik zu entwerfen, die Räder der falschen Größe bauen wird.

RPG-Programmierer haben ein Rad, schade, daß es viereckig ist.

SYSTEM-Analytiker sind viel zu beschäftigt, Räder zu suchen, um eins zu bauen.

TEXTVERARBEITUNGS-Benutzer bauen Räder.

TOS-Benutzer bleiben wo sie sind, sie sind das Warten gewohnt.

UNIX hat unter irgendeiner Schale sicher irgendwo ein Verzeichnis von Rädern.