2.4. Felder (arrays)

Wiederholung: arrays (auf deutsch: Felder) beschreiben n-Tupel über einer Menge, also Folgen über dem gleichen Datentyp. Der Zugriff auf die einzelnen Komponenten erfolgt direkt über einen Index. Der Wert von n kann flexibel sein; der Indexbereich ist ein zusammenhängender Subtyp eines geordneten Datentyps. Grundsätzliche Form:

type <name> **is array** <Indexbereich> **of** <Datentyp>

Beispiele:
type Hvektor **is array** (1..100) **of** float;
type Hmatrix **is array** (1..50) **of** Hvektor;
type Bundesligatabelle **is array** (1..18) **of** fussballverein;
type codierung **is array** (Character) **of** Character;

Die iterierte array-Bildung kürzt man ab, indem alle Indexbereiche in eine Definition geschrieben werden:

Statt
type Hvektor **is array** (1..100) **of** float;
type Hmatrix **is array** (1..50) **of** Hvektor;
type Hvolume **is array** (1..80) **of** Hmatrix;

schreibt man also kurz:
type Hvolume **is array** (1..100,1..50,1..80) **of** float;

Die Zahl der hierbei verwendeten Indexbereiche heißt die *Dimension* des Feldes. Hvolume ist also ein 3-dimensionales Feld.

Statt Konstanten dürfen in den Indexbereichen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann.

Beispiele (wobei f und g Funktionen vom Ergebnistyp integer sein sollen):
type Hvektor **is array** (1..N, x..I*J) **of** Boolean;
type ausschnitt **is array** (f(unten)..g(oben)) **of** integer;
type sonstiges **is array** ((oben-unten) div 2..f(g(unten*oben))) **of** float;

Ein Feld heißt *statisch*, wenn zur Übersetzungszeit die Feldgrenzen alle bekannt sind. Wird mindestens eine Feldgrenze erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch*.

Auch *unspezifizierte* Feldgrenzen sind zulässig. Man trägt dann nur den Datentyp des Index ein und fügt ein
range <> ("<>" sprich "box")
hinzu. Solche unspezifizierten array-Deklarationen muss man bei ihrer Verwendung spezifizieren, z.B. bei der Deklaration von Variablen und beim konkreten Parameteraufruf.

Beispiele:
type text **is array** (natural **range** <>) **of** Character;
type raster **is array** (integer **range** <>, integer **range** <>) **of** pixel;
type ganzzahlvektor **is array** (integer **range** <>) **of** integer;
procedure Sort (A: **in out** ganzzahlvektor; unten, oben: integer) **is** ...

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch ein, z.B.:
X: ganzzahlvektor (-10..10) oder Y: ganzzahlvektor (I..J)

Ein einfaches Problem ist die Einbettung mehrdimensionaler Felder in eindimensionale. Diese Aufgabe muss jeder Compiler lösen können, da heutige Speicher in der Regel eindimensional sind. Die übliche Einbettung lautet:

Wenn das n-dimensionale Feld ($n \geq 2$) von der Form

array ($u_1..o_1$, $u_2..o_2$, ..., $u_n..o_n$) **of** ...

und das eindimensionale Feld von der Form

array (unten..oben) **of** ...

und alle Indextypen ($u_j..o_j$) und (unten..oben) ganzzahlige Intervalle sind, dann kann man den Index (i_1, i_2, ..., i_n) abbilden auf

$$f(i_1, i_2, ..., i_n) = unten + \sum_{k=1}^{n} d_k \cdot (i_k - u_k) \quad \text{mit} \quad d_k = \prod_{j=k+1}^{n} (o_j - u_j + 1)$$

Nebenbedingung: oben-unten+1 = d_0. (beachte: $d_n = 1$)

f heißt Speicherabbildungsfunktion.

Hinweis 1: Wenn ein Compiler ein mehrdimensionales Feld in die Maschinensprache übersetzt, dann legt er zugleich ein Feld für die Werte d_k an, k = n, n-1, ..., 1. Bei jedem Zugriff auf das Feld wird dann $f(i_1, i_2, ..., i_n)$ berechnet.

Ein "optimierender" Compiler wird bei **for**-Schleifen die Differenz zwischen den Indices zweier aufeinander folgender Zugriffe ermitteln und versuchen, das Fortschalten des Index durch Addition eines Differenzterms zu erledigen, so dass f nur beim ersten Mal berechnet wird und anschließend nicht mehr. Allerdings wird hierbei eine eventuelle Bereichsüberschreitung bei den Grenzen nicht erkannt, weshalb der Compiler mehrere Möglichkeiten bei der Übersetzung hat, die i.A. der Benutzer auswählen kann (in Ada: pragma).

Hinweis 2: Die oben angegebene Funktion f speichert das mehrdimensionale Feld "zeilenweise" (machen Sie sich dies an einer Matrix klar!).

Will man spaltenweise speichern (FORTRAN macht das so), dann muss man die Funktion leicht modifizieren. (Wie?)

Hinweis 3: Die Funktion f lässt sich auch schreiben als:

$$f(i_1, i_2, ..., i_n) = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k + \sum_{k=1}^{n} d_k \cdot i_k = u_{reduz} + \sum_{k=1}^{n} d_k \cdot i_k$$

mit der "reduzierten Anfangsadresse" $u_{reduz} = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k$

Ein Compiler speichert daher u_{reduz} und die d_k-Werte sowie eventuell alle o_j und u_j, um die Bereichsüberschreitung zu testen.

Beispiel 2.1: Intervallschachtelung

Ein Feld A: **array** (1..n) **of** integer sei gegeben. Das Feld sei sortiert, d.h.: $A(i) \leq A(i+1)$ für $i = 1, 2, ..., n-1$.

Aufgabe: Man schreibe einen Algorithmus, der zu einer Zahl s in möglichst kurzer Zeit feststellt, ob s im Feld A liegt oder nicht. Im Falle, dass s im Feld A enthalten ist, soll ein Index m mit $A(m) = s$ ausgegeben werden, anderenfalls sei $m = 0$.

Geht man das Feld von links nach rechts durch, so dauert es bis zu n Schritte, um das Ergebnis zu ermitteln. Ein schnelleres Verfahren ist die bekannte Intervallschachtelung: Teste, ob s genau in der Mitte A(mitte) von A liegt, falls nein und ist $A(\text{mitte}) < s$ ist, suche rechts von der Mitte weiter, sonst links.

Vergleiche Manuskript Plödereder, Abschnitt 4.1.2; dort heißt die Intervallschachtelung "Binäre Suche".

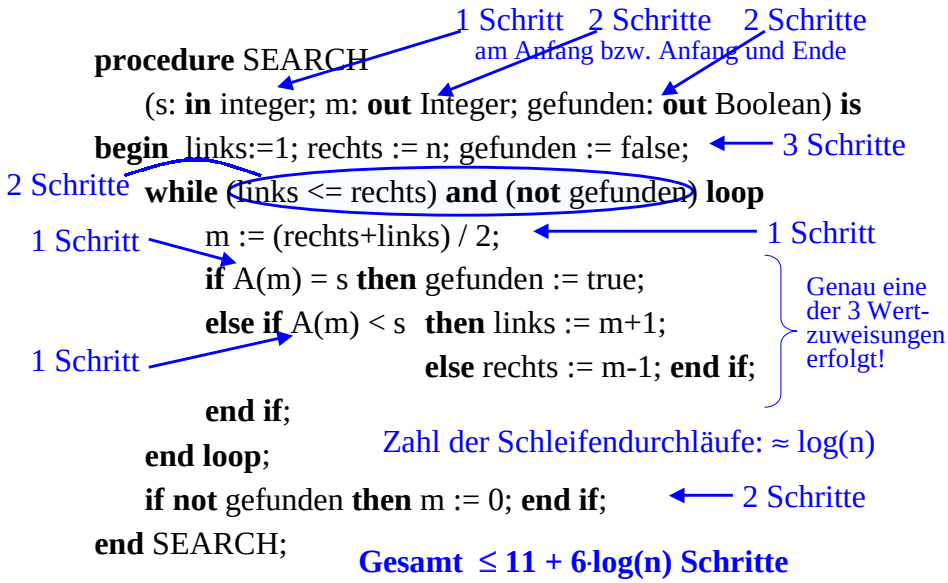
Programm 1: Intervallschachtelung oder binäre Suche

```
procedure SEARCH
        (s: in integer; m: out Integer; gefunden: out Boolean) is
procedure Intervallschachtelung (links, rechts: in integer) is
begin
    if links <= rechts then
        m := (rechts+links) / 2;
        if A(m) = s then gefunden := true;
        else if A(m) < s  then Intervallschachtelung (m+1,rechts);
                          else Intervallschachtelung (links, m-1); end if;
    else gefunden := false; m:=0;
    end if;
end Intervallschachtelung;
begin  Intervallschachtelung (1,n)  end SEARCH;
```


Programm 2: Man kann auch eine iterative Version angeben:

```
procedure SEARCH
    (s: in integer; m: out Integer; gefunden: out Boolean) is
begin  links:=1; rechts := n; gefunden := false;
    while (links <= rechts) and (not gefunden) loop
        m := (rechts+links) / 2;
        if A(m) = s then gefunden := true;
        else if A(m) < s  then links := m+1;
                          else rechts := m-1; end if;
        end if;
    end loop;
    if not gefunden then m := 0; end if;
end SEARCH;
```

Aufwand (für beide Programme), uniforme Zeitkomplexität:



Der schlechteste Fall kann auch tatsächlich eintreten, wenn nämlich das gesuchte Element s nicht im Feld A enthalten ist. Die *uniforme worst case time-complexity* lautet daher

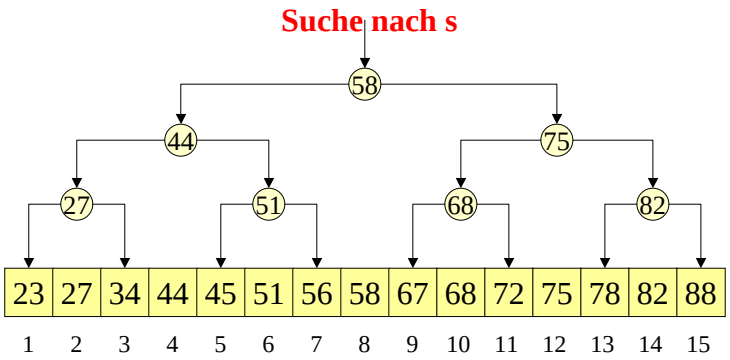
$t(n) = 11 + 6 \cdot \log(n)$.

Beachten Sie: n ist hier nicht die Länge der Eingabe, sondern die Anzahl der Elemente im Feld A.

Was ist der beste Fall? In diesem Fall wird s im ersten Durchgang der while-Schleife gefunden, d.h. nach 17 Schritten ist die Prozedur beendet.

Mit wievielen Schritten muss man im Mittel rechnen? Hierzu nehmen wir an, dass sich das gesuchte Element s tatsächlich im Feld A befindet (sonst kann man nur die worst case Abschätzung verwenden).

Wir skizzieren die Verhältnisse, wobei wir hier n=15=2^4-1 wählen:



In 2^3 = 8 Fällen braucht man 4 Schleifendurchläufe,
in 2^2 = 4 Fällen braucht man 3 Schleifendurchläufe,
in 2^1 = 2 Fällen braucht man 2 Schleifendurchläufe,
in 2^0 = 1 Fall braucht man 1 Schleifendurchlauf.

Allgemein gilt also, wenn $n = 2^k - 1$ ist:

In 2^{k-1} Fällen braucht man k Schleifendurchläufe,
in 2^{k-2} Fällen braucht man k-1 Schleifendurchläufe,
in 2^{k-3} Fällen braucht man k-2 Schleifendurchläufe,
....
in 2^0 = 1 Fall braucht man 1 Schleifendurchlauf.

Daher braucht man im Mittel:

$$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + (k-2)\cdot 2^{k-3} + \ldots + 2\cdot 2^1 + 1) \text{ Durchläufe.}$$

Berechne also die Summe $\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = \frac{1}{2}\sum_{j=1}^{k} (j-1)\cdot 2^{j} + \frac{1}{2}\sum_{j=1}^{k} 2^{j}$$

$$= \sum_{j=1}^{k} (j-1)\cdot 2^{j-1} + \frac{1}{2}(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^{j} + (2^{k} - 1) = \sum_{j=1}^{k} j\cdot 2^{j} - k\cdot 2^{k} + (2^{k} - 1),$$ d.h.:

$\frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = k\cdot 2^{k} - (2^{k} - 1)$. Folglich erhalten wir

$$\frac{1}{n}(k\cdot 2^{k-1}+(k-1)\cdot 2^{k-2}+\ldots+2\cdot 2^{1}+1) = \frac{k\cdot 2^{k} - (2^{k} - 1)}{2^{k} - 1} \approx k - 1$$

Somit beträgt die *average case time-complexity* der Intervallschachtelung ziemlich genau **11 + 6·(log(n)-1)** Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

Erkenntnis: Im Mittel spart man nur eine konstante Zahl an Operationen gegenüber dem schlechtesten Fall. Folglich lohnt sich zum Beispiel die Abfrage " A(m) = s " nicht; könnte man sie weglassen, so würde man log(n) viele Schritte sparen. Dies führt auf folgende bessere Version des Algorithmus für die Suche mittels Intervallschachtelung: Man entscheide erst ganz am Ende, ob A(m) = s gewesen ist; hierzu muss man im Falle A(m) < s in dem rechten Teil des Feldes weitersuchen (links:=m+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes m (rechts:=m).

Programm 3: Verbesserte iterative Version:

```
procedure SEARCH
    (s: in integer; m: out Integer; gefunden: out Boolean) is
begin  links:=1; rechts := n;
    while (links < rechts) loop
        m := (rechts+links) / 2;
        if A(m) < s then links := m+1;
                    else rechts := m; end if;
        end if;
    end loop;
    gefunden := A(m) = s;
    if not gefunden then m := 0; end if;
end SEARCH;
```


Weisen Sie nun nach, dass für diese Version gilt:

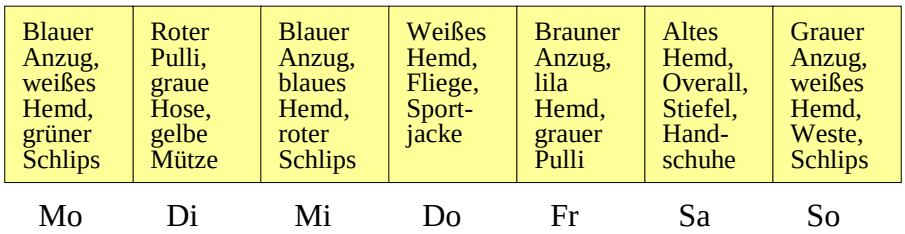
Die *uniforme worst case time-complexity* beträgt $11 + 4 \cdot \log(n)$; die *uniforme average case time-complexity* besitzt nun genau den gleichen Wert, da ja die Entscheidung, ob A(m)=s ist, in jedem Fall erst nach dem log(n)-maligen Durchlaufen der Schleife gefällt wird!

Hierbei ist n die Zahl der Elemente im array.

Weitere Suchverfahren auf Feldern betrachten wir in Kapitel 4.

Wie kann man Felder nutzen? Hier sind drei Möglichkeiten:

1. Direkte Speicherung der Information, also: das Feld als Speicher - und Suchstruktur:



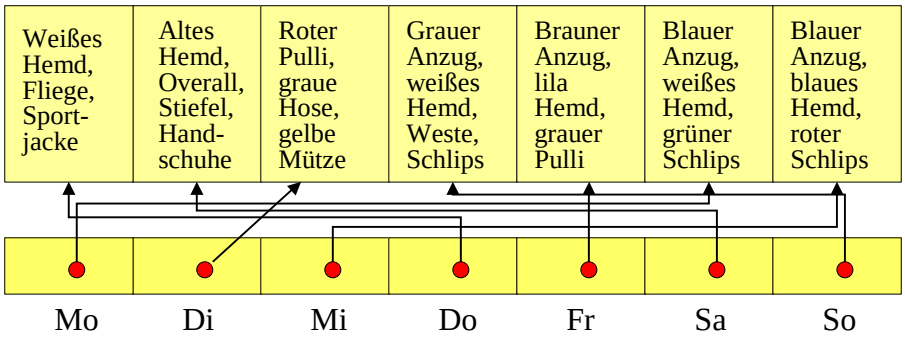
type wochentag **is** (Mo, Di, Mi, Do, Fr, Sa, So);

type kleidung **is array** (1..100) **of** Character;

type anziehen **is array** (wochentag) **of** kleidung;

2. Nur Speicherung der Adressen, also nur als Suchstruktur:

In der Halde können die Daten beliebig irgendwo liegen:



type kleidung **is array** (1..100) **of** Character;
type zugriff **is access** kleidung;
type anziehen **is array** (wochentag) **of** zugriff;

3. Getrennte Felder für Speicherung und Sortieren/Suche:

Weißes Hemd, Fliege, Sport-jacke	Altes Hemd, Overall, Stiefel, Hand-schuhe	Roter Pulli, graue Hose, gelbe Mütze	Grauer Anzug, weißes Hemd, Weste, Schlips	Brauner Anzug, lila Hemd, grauer Pulli	Blauer Anzug, weißes Hemd, grüner Schlips	Blauer Anzug, blaues Hemd, roter Schlips
1	2	3	4	5	6	7

6	3	7	1	5	2	4
Mo	Di	Mi	Do	Fr	Sa	So

```
type kleidung is array (1..100) of Character;
type index is range 1..7;
type anziehsachen is array (index) of kleidung;   -- Speicherung
type sortierfeld is array (wochentag) of index;   -- Zugriff
```


Beispiel 2.2: Noch ein kurzes Standardbeispiel:
Codieren durch Buchstabenverschiebung um N Stellen:

```
type Codierung is array (Character) of Character;
X: Codierung; N: Natural; A: Character;

procedure EinsVerschieben is
begin  for A in Character loop
          if X(A) = Character'Last then X(A):=Character'First
          else X(A) := succ(X(A)); end if; end loop;
end;     .....

for A in Character loop X(A) := A; end loop;
for I in (1..N) loop EinsVerschieben; end loop;   ....
--  Die Codierung durch Verschieben um N Stellen im Alphabet
    --    erfolgt anschließend durch:
get (A); put(X(A)); ...
```

Bemerkung: Wir verwenden öfters den Logarithmus log(n) wie eine ganze Zahl. Was genau ist damit gemeint?

Mathematische Definition: für $x > 0$, $b > 1$ gilt
$\log_b(x) = y \Leftrightarrow x = b^y$.

Der ganzzahlige Anteil des Logarithmus von x zur Basis b ist für x>1 im Wesentlichen die Länge der Zahlendarstellung von x im Stellenwertsystem zur Basis b.

Beispiele (man schreibt auch lg, ld, ln bei Basis 10, 2 bzw. e = 2,7182818...):
$\log_{10}(17635) = \lg(17635) = 4{,}2464... \approx 5 = \lceil \log_{10}(17635) \rceil$
$\log_2(3{,}8) = \mathrm{ld}(3{,}8) = 1{,}9260... \approx 2 = \lceil \log_2(3{,}8) \rceil$
$\log_e(8{,}2) = \ln(8{,}2) = 2{,}1041... \approx 3 = \lceil \log_e(8{,}2) \rceil$

Für uns ist der Logarithmus log die Länge der Darstellung zur jeweiligen Basis, wobei wir stets die Basis 2 annehmen, sofern nichts anderes gesagt wird. Also definieren wir:

Unsere Definition: log: $\mathbb{N}_0 \to \mathbb{N}_0$ für alle $n > 0$ definiert durch
$\log(n) = y \Leftrightarrow$ y ist die kleinste natürliche Zahl mit $n < b^y$.
Weiterhin sei ist $\log(0) = 1$. Einige Werte:

n	log(n)	n	log(n)	n	log(n)
0	1	8	4	128	8
1	1	9	4	500	9
2	2	10	4	512	10
3	2	15	4	1000	10
4	3	16	5	9000	14
5	3	31	5	10^6	20
6	3	32	6	10^9	30
7	3	64	7	10^{12}	40

Hinweis 1: Wenn $\log_2$ der mathematisch exakt definierte reellwertige Logarithmus ist, dann gilt $\log(0) = 1$ und für alle $n > 0$: $\log(n) = \lceil \log_2(n+1) \rceil$.
Da sich unser Logarithmus und der exakte Logarithmus für $n \geq 1$ immer höchstens um 1 unterscheiden, werden wir diese beiden Funktionen als "im Wesentlichen gleich" auffassen.

Hinweis 2: Beachten Sie, dass sich die Logarithmen zu zwei verschiedenen Basen a und b immer nur um die Konstante $\log_a(b)$ unterscheiden,
denn es gilt mathematisch für alle $x>0$:

$$\log_a(x) = \log_b(x) \cdot \log_a(b).$$

Speziell gilt daher für alle Basen $a>1$ und $b>1$:

$$O(\log_a(n)) = O(\log_b(n)) = O(\log(n)).$$