Kapitola 4

Algoritmizace nad strukturovanými proměnnými

4.1 Algoritmy nad polem

V algoritmu zavádíme proměnnou typu pole v případě, že potřebujeme opakovaně pracovat s i-tou složkou vstupní posloupnosti hodnot. Tak rozšíříme-li zadání Příkladu 2.2 o potřebu zjistit počet pracovníků v podniku s nadprůměrným platem, je zapotřebí v souladu s uvedeným řešením opět zjistit průměrný plat v podniku. Až přečteme poslední hodnotu vstupní posloupnosti, je možno vypočítat průměrný plat. V tom okamžiku máme v operační paměti proměnné s následujícími obsahy:

SUMA	16000	POCET	5	PLAT	0	AP	3200.0

Nyní je však třeba vypočtený průměr porovnávat postupně s prvním, druhým ... vstupujícím platem a zjišťovat, zda je či není možno provádět potřebný příkaz k určení počtu zaměstnanců s NADprůměrným platem (jenž je tedy podmíněn, k čemuž použijeme podmíněný příkaz):

if PLAT[I] > AP **then** NAD:=NAD +1

Je tedy zřejmé, že proměnná PLAT nemůže být proměnnou jednoduchého typu (tj. místem pro uložení elementární hodnoty - jednoho čísla), ale musí být proměnnou strukturovaného typu, typu pole. Jednotlivé složky takovéto proměnné rozlišíme pomocí indexu - zápisem indexované proměnné.

Řešení za předpokladu, že na vstupu bude méně, maximálně však 30 (obecně N) platů zaměstnanců, je ve tvaru programu:

```
Program  NADPRUMER;
   const N    = 30;      NPLUS1   = 31;
   type  POLE = array[1..NPLUS1] of word;
   var   POCET,I,NAD : byte;      PLAT : POLE;
         SUMA        : longint;   AP   : real;
   begin SUMA:=0;      POCET:=0;         I:=1;
         write('Zadej plat:');
         readln(PLAT[I]);
         while PLAT[I] <> 0 do begin  SUMA:=SUMA + PLAT[I];
                                      POCET:=POCET +1;  I:=I+1;
                                      write('Zadej další plat:');
                                      readln(PLAT[I])
                               end;
         AP  := SUMA/POCET;
         NAD := 0;
         for I:=1 to POCET do if PLAT[I] > AP then NAD:=NAD+1;
         writeln('Počet nadprůměrných platů je :',NAD)
   end.
```

Prostor pro ukládání platů musí být dostatečně velký, tj. musí v něm být místo pro maximálně 30 platů + jedno místo pro koncovou hodnotu.

Jak jsme řekli už v úvodu, k řešení každého problému existuje značné množství algoritmů. Je prakticky nemožné určit, zda právě naše řešení je optimální. Ve výše uvedeném programu např. vidíme, že příkazy

POCET:=POCET+1 a I:=I+1

provádějí stejnou operaci a zřejmě jeden z nich by bylo možno vypustit, přestože na počátku jsou proměnné POCET a I definovány různými hodnotami, neboť jsou použity příkazy

POCET:=0 a I:=1.

Vynecháme-li v programu příkazy např.

POCET:=0 a POCET:=POCET+1

a před příkaz výpočtu aritmetického průměru zařadíme příkaz POCET:=I–1
dostáváme stejně správné řešení daného problému, jako když vynecháme příkazy

I:=1 a I:=I+1

a na začátku příkazové části nahradíme příkaz

POCET:=0 příkazem POCET:=1,

a v těle cyklu nahradíme index I indexem POCET a dále uvedeme příkazy ve tvaru

AP:=SUMA/(POCET - 1);
for I:=1 **to** POCET - 1 **do if** PLAT[I]>AP **then** NAD:=NAD+1.

Troufne si čtenář určit, která ze tří možných modifikací programu je optimální? Není to náhodou nějaký úplně jiný algoritmus?

Příklad 4.1.1: Na vstupu jsou dvě setříděné sady celočíselných hodnot. Každá z obou vstupních posloupností je ukončena dohodnutým koncovým číslem např. –999. Vytvořte jednu proměnnou V obsahující vzestupně setříděné hodnoty z obou vstupních posloupností.

Tvar vstupů může být např.: $a_1\ a_2 \ldots a_i \ldots a_m\ -999$
$b_1\ b_2 \ldots b_i \ldots a_n\ -999$, tj.
počet členů jednotlivých posloupností není předem dán a obecně je m $\neq$ n.

Pro část definicí a deklarací je třeba vyjasnit předpokládaný počet členů v jednotlivých vstupních posloupnostech. Předpokládejme, že m i n bude maximálně 20. Pak můžeme část definicí a deklarací napsat ve tvaru:

```
type INDEX1 = 1..20;         INDEX2 = 1..40;
     VSTUP  = array[INDEX1] of integer;
     VYSTUP = array[INDEX2] of integer;
 var A,B    : VSTUP;         V      : VYSTUP;
     M,N,XY : INDEX1;        K,Z    : INDEX2;
```

Načtení vstupů provedeme příkazy např.: popř. algoritmicky lépe příkazy:

```
M:=1;  read(A[M]);
while A[M]<>-999 do begin M:=M+1;
                          read(A[M])
                    end;
M:=M-1;
```

```
M:=0; read(POM)
while POM <> -999 do begin M:=M+1;
                           A[M] := POM;
                           read(POM)
                     end;
```

a tytéž příkazy je možno použít ještě jednou se záměnou N za M a B za A. Již však víme, že tam, kde by měla začít manuální práce na algoritmu, musí místo toho nastoupit myšlení, začít strukturované programování. K načtení jedné posloupnosti vytvoříme podprogram:

```
Procedure CTIVEKTOR(var VEKTOR:VSTUP; var POCET:INDEX1);
    var   POM : integer;
    begin writeln('Zadávej prvky vektoru (konec při -999): ');
          POCET:=0;
          write('Zadávej ',POCET+1,'.prvek: ');readln(POM);
```

```
        while POM <> -999 do
              begin POCET:=POCET+1;
                    VEKTOR[POCET]:=POM;
                    write('Zadej ',POCET+1,'.prvek: ');
                    readln(POM);
              end;
  end;
```

a tento podprogram, do kterého jsme zavedli dialog, hned na začátku příkazové části programu dvakrát použijeme: CTIVEKTOR(A,M);
CTIVEKTOR(B,N);

Jádrem řešeného algoritmu je zpracování proměnných A a B a definování obsahu proměnné V. Práci máme usnadněnou v tom, že hodnoty načítané do proměnné A resp. B byly již na vstupu setříděny. Kdyby tomu tak nebylo, pak bychom použili některou z uvedených procedur pro setřídění řady čísel v Příkladu 4.1.3, např.

BUBBLE(A,M); resp. BUBBLE(B,N).

Příkazy vedoucí k požadovanému vzniku obsahu proměnné V pak mohou být ve tvaru:

```
X:= 1;   Y:= 1;   Z:= 1;
while (X<=M) and (Y<=N) do
      begin if A[X]<B[Y] then begin V[Z]:=A[X];
                                    X:=X+1
                              end
                         else begin V[Z]:=B[Y];
                                    Y:=Y+1
                               end;
            Z:=Z+1
      end;             { odcházíme z cyklu, pakliže je     }
                       { jeden ze vstupních vektorů vybrán }
while X<=M do begin V[Z]:=A[X];
                    X:=X+1;  Z:=Z+1
                end;     { případně jsme přepsali konec  }
                         { vektoru A do vektoru V        }
while Y<=N do begin V[Z]:=B[Y];
                    Y:=Y+1;  Z:=Z+1
                end;     { případně jsme přepsali konec }
                         { vektoru B do vektoru V       }
```

Poslední dva uvedené cykly **while–do** s výhodou nahradíme dvojící cyklů **for–to–do** (za předpokladu deklarace **var** I:INDEX2) ve tvaru:

```
for I:=X to M do V[Z-X+I] := A[I];
for I:=Y to N do V[Z-Y+I] := B[I]
```

Poněvadž přehledný tisk výsledků by měl zřejmě obsahovat jak hodnoty vstupních vektorů, tak hodnoty výstupního vektoru, je výhodné opět deklarovat proceduru:

Procedure TISKVEKTOR(VEKTOR : VSTUP / ? / VYSTUP ; POCET: INDEX1 / ? / INDEX2);

var POM : INDEX1 / ? / INDEX2 ;

begin for POM:=1 **to** POCET **do**
if (POM **mod** 20)=0 **then** writeln(VEKTOR[POM]:4)
else write(VEKTOR[POM]:4);
writeln; writeln;
end;

a pak ji použít v příkazech výstupu, např.:

```
writeln('Vstupní vektor A je: ');
TISKVEKTOR(A,M);
writeln('Vstupní vektor B je: ');
```

TISKVEKTOR(B,N);
writeln('Výstupní vektor V je: ');
TISKVEKTOR(V,Z–1);

Problém, jak již byl naznačen při výše uvedené deklaraci procedury TISKVEKTOR, je v požadavku, aby typ formálního a skutečného parametru byl identický (totožný). Poněvadž však máme definovaný jiný typ vstupních vektorů A,B a jiný typ výstupního vektoru V, nelze pro jejich tisk použít stejného podprogramu. Máme tedy dvě možnosti:

- buď deklarovat proměnné A,B a V na základě stejného typu VYSTUP s tím, že řada složek vektoru A a B nebude nikdy obsazena (definována) – vědomě plýtváme s pamětí
- nebo napsat dva různé podprogramy.
- nebo použít jiné strategie algoritmizace problému

Rozhodnutí záleží vždy na konkrétním zadání. Ve školních příkladech, kde pracujeme s poměrně malými datovými strukturami, se zřejmě rozhodneme pro první možnost. Část definicí programu pak bude mít tvar:

```
type  INDEX = 1..40;
      POLE = array[INDEX] of integer;
var   A,B,V : POLE;
      M,N,K,X,Y,Z : INDEX;
```

a část definicí a deklarací včetně hlavičky procedury bude mít tvar:

```
Procedure  TISKVEKTOR(VEKTOR:POLE;POCET:INDEX);
    var    POM:INDEX;
```

s obdobnou změnou v hlavičce procedury CTIVEKTOR na tvar:

```
Procedure CTIVEKTOR(var VEKTOR:POLE; var POCET:INDEX);
```

konec Příkladu 4.1.1 konec Příkladu 4.1.1

Příklad 4.1.2: Na vstupu je několik (maximálně však 25) celočíselných hodnot ukončených hodnotou −999. Vytvořte dvě proměnné L a S tak, aby v proměnné L byly všechny liché hodnoty ze vstupu a v proměnné S byly všechny sudé hodnoty ze vstupu. Obsahy proměnných L a S přehledně vytiskněte.

S využitím znalostí, získaných algoritmizací problému uvedeného v Příkladu 4.1.1, můžeme napsat výslednou podobu programu.

```
Program  ROZDEL;
    type INDEX = 1..25;
         POLE  = array[INDEX] of integer;
    var  A,L,S : POLE;
         N,PL,PS,I : INDEX;
    Procedure CTIVEKTOR(var VEKTOR:POLE;var POCET:INDEX);
        begin writeln('Zadávej prvky vektoru (konec při -999): ');
              POCET:=1;
              write('Zadávej ',POCET,'.prvek: ');readln(VEKTOR[POCET]);
              while VEKTOR[POCET]<>-999 do
                    begin POCET:=POCET+1;
                          write('Zadej ',POCET,'.prvek: ');
                          readln(VEKTOR[POCET]);
                    end;
              POCET:=POCET-1
        end;
    Procedure TISKVEKTOR(VEKTOR:POLE;POCET:INDEX);
        var   POM:INDEX;
        begin for POM:=1 to POCET do
                  if(POM mod 20)=0 then writeln(VEKTOR[POM]:4)
                                   else write(VEKTOR[POM]:4);
              writeln; writeln
```

```
        end;
    begin CTIVEKTOR(A,N);
          PL:=1; PS:=1;
          for I:=1 to N do
              if odd(A[I]) then begin L[PL]:=A[I];
                                      PL:=PL+1
                                end
                           else begin S[PS]:=A[I];
                                      PS:=PS+1
                                end;
          writeln('Vstupní vektor čísel je: ');  TISKVEKTOR(A,N);
          writeln('Vektor lichých čísel je: ');  TISKVEKTOR(L,PL-1);
          writeln('Vektor sudých čísel je: ');   TISKVEKTOR(S,PS-1)
    end.
```

Zápis procedury CTIVEKTOR není tak algoritmicky čistý, jak bylo uvedeno v Příkladě 4.1.1 (srovnej). Často se však s uvedeným nedostatkem setkáváme. Je třeba si uvědomit možné těžkosti a v našem případě je třeba přinejmenším definovat typ INDEX jako INDEX=1..26.

konec Příkladu 4.1.2 konec Příkladu 4.1.2

Příklad 4.1.3: Je dán vektor N celých čísel (desetinných čísel, řetězců znaků (např. příjmení), znaků atp.). Vytvořte proceduru pro jejich seřazení od nejmenší po největší.

Řešení : Za předpokladu, že jsou definovány typy

type INDEX = 0..MAXPOCET;
SLOZKA = integer; {může být real, **string**[20] atp.}
A = array[INDEX] of SLOZKA;
kde MAXPOCET je definovaná konstanta např.
const MAXPOCET=20;
omezující použitelnost dále uvedených procedur
na zpracování N-členných vektorů
(N je maximálně MAXPOCET).

K setřídění položek vektoru použijeme metodu přímého výběru (straight selection).

Princip: - ze všech položek vektoru vyber nejmenší hodnotu
- vyměň vzájemně nalezenou hodnotu s hodnotou v prvé položce vektoru.
- poté vyber nejmenší prvek ze zbylých N-1 prvků (z 2. až N-té položky) a vyměň ho s druhou položkou atd. až zůstane poslední N-tý (maximální) prvek.

```
procedure STRSEL(var POLE:A; N:INDEX);
    var I,J,K : INDEX;   POM:SLOZKA;
    begin for I:=1  to N-1 do
              begin K:=I;  POM:=POLE[I];
                    for J:= I+1  to N do if POLE[J]<POM then
                                            begin K:=J;
                                                  POM := POLE[J]
                                            end;
                    POLE[K] := POLE[I];
                    POLE[I] := POM
              end
    end;
```

K setřídění položek vektoru použijeme metodu přímého vkládání (straight insertion).

Princip : V i-tém kroku se zpracovává i-tá položka. Buď se ponechá na svém původním místě (je-li větší jak i-1. položka), nebo se vloží na správné místo. Tomu předchází uvolnění místa příslušné položky posunutím hodnot, počínaje i-1. položkou a konče příslušnou položkou, vždy o jedno místo vpravo.

```
procedure STRINS(var POLE:A; N:INDEX);
     var I,J :INDEX; POM :SLOZKA;
     begin for I:=2 to N do
               begin POM :=POLE[I]; POLE[0] :=POM;
```

```
                J :=I-1;
                while POM<POLE[J] do
                      begin POLE[J+1] :=POLE[J];
                            J:= J-1
                      end;
                POLE[J+1] := POM  { též POLE[I] :=POM }
            end
    end;
```

K setřídění položek vektoru použijeme některou z metod přímé výměny. Metody jsou založeny na principu srovnávání a případné výměny sousedních položek vektoru tak dlouho, dokud není vektor seřazený

a) bublinové třídění - Bubble sort

Princip : V i-tém kroku nám probublá i-tá nejmenší hodnota na i-té místo od začátku (vrcholu) vektoru.

V každém kroku porovnáváme všechny položky vektoru po sousedních dvojicích. Je-li prvý prvek dvojice větší než druhý, provedeme výměnu. Po N-1 krocích máme jistotu, že vektor je setříděný.

```
procedure BUBBLE(var POLE:A; N:INDEX);
    var I,J :INDEX;  POM :SLOZKA;
    begin for I:=2  to N  do
              begin for J:=N downto 2 { s výhodou downto I }
                        do if POLE[J-1]>POLE[J] then
                              begin POM := POLE[J-1];
                                    POLE[J-1] := POLE[J];
                                    POLE[J] := POM
                              end
              end
    end;
```

b) u bublinového třídění s výhodou (tzv. Ripple sort) je vzata v úvahu skutečnost, že vektor může být setříděný dříve než po N-1 krocích. Testujeme, zda v I-tém kroku došlo k výměně alespoň u jedné dvojice. Pokud ne, je již vektor setříděn.

```
        procedure RIPPLE(var POLE:A; N:INDEX);
            var I,POC : INDEX;
                POM : SLOZKA;
                SETRIDENO : Boolean;
            begin POC := N-1;
                  repeat SETRIDENO := true; { setříděno }
                         for I:=1 to POC do
                             if POLE[I+1]<POLE[I] then
                                begin POM:= POLE[I];
                                      POLE[I]:= POLE[I+1];
                                      POLE[I+1] := POM;
                                      SETRIDENO := false
                                end;
                         POC := POC-1
                  until (POC = 0) or SETRIDENO
            end;
```

c) Další modifikací bublinové metody je Shaker sort. V určitém kroku prohlížíme řazený vektor dvakrát. Nejprve např. od poslední (dolní) D. dvojice po horní H. dvojici. Zapamatujeme si číslo J–té poslední dvojice, u které jsme byli nuceni provést výměnu. Vzápětí prohlížíme vektor podruhé, tentokráte shora (od H = J+1. dvojice) dolů (po poslední dvojici (D)). Opět si zapamatujeme místo poslední výměny (I=J) a jdeme totéž opakovat do dalšího kroku.

Seřazení vektoru nastává v okamžiku, kdy ani při průchodu zdola nahoru, ani při průchodu shora dolů neprovádíme výměnu.

```
Procedure SHAKER(var POLE:A; N:INDEX);
```

```
    var I,J,H,D : INDEX;
        POM     : SLOZKA;
    begin H := 2;     D := N;     I := N;
          repeat for J := D downto H do
                    if POLE[J-1] > POLE[J] then
                       begin POM := POLE[J-1];
                             POLE[J-1] := POLE[J];
                             POLE[J] := POM;
                             I := J
                       end;
                 H := I + 1;
                 for J:= H to D do
                    if POLE[J-1] > POLE[J] then
                       begin POM := POLE[J-1];
                             POLE[J-1] := POLE[J];
                             POLE[J] := POM;
                             I := J
                       end;
                    D := I-1;
          until H > D;
end;
```

Předchozí tři metody (Bubble, Ripple sort a Shaker sort) vycházely z principu, že v každém kroku se prohlédly všechny dvojice sousedních položek vektoru. Poslední z ukázaných jednoduchých metod řazení, metoda Shuttle sort, vychází opět z prohlížení sousedních položek tříděného vektoru, avšak jakmile je zjištěna nutnost provedení výměny, je provedena a vektor je opět prohlížen od prvých dvou položek.

```
procedure SHUTT(var POLE:A; N:INDEX);
    var I :INDEX;
        POM :INTEGER;
    begin I:=2;
          while I<=N do if POLE[I]<POLE[I-1] then begin POM:=POLE[I];
                                                        POLE[I]:=POLE[I-1];
                                                        POLE[I-1]:=POM;
                                                        I:=2
                                                  end
                                             else I :=I+1
    end;
```

K seřazení většího množství dat se používají duchaplnější metody, z nichž jmenujme alespoň Quick sort, Shell sort a Heap sort. Prvním dvěma uvedeným se ještě trochu věnujme:
Metoda Quick sort vychází z následujícího principu:
Z vektoru se vyhledá vhodná hodnota POM1 (např. hodnota umístěná uprostřed). Pak se prochází vektor řazených hodnot zleva, až se najde číslo větší než POM1, načež se prohlíží vektor zprava, až se najde hodnota menší než POM1. Tato nalezená čísla se vzájemně vymění. Uvedený postup aplikujeme tak dlouho, dokud se výměna nedá provést a vektor je rozdělen na dvě části (např. poloviny); v levé polovině jsou čísla menší než nějaká hodnota POM1, v pravé polovině pak jsou čísla větší než nějaká hodnota POM1. Celý proces opakujeme s každou polovinou hodnot samostatně (dále čtvrtinou atd.), až nakonec seřadíme sousední dvojice a tím je úloha vyřešena. Popsaný algoritmus obsahují následující dva podprogramy, z nichž druhý je rekurzivní podprogram.

```
   Procedure QUICK(var POLE:A; N: INDEX);
        const M = 12;
        type STRUKTURA = array[1..M] of record H, D : INDEX
                                        end;
        var I, J, H, D : INDEX;       S : 0..M;
            POM1, POM2 : SLOZKA;      Z : STRUKTURA;
        begin S := 1;   Z[1].H := 1;   Z[1].D := N;
              repeat H := Z[S].H;   D := Z[S].D;   S := S-1;
                     repeat I := H;   J := D;   POM1 := POLE[(H+D) div 2];
```

```
                        repeat while POLE[I] < POM1 do I := I+1;
                               while POM1 < POLE[J] do J := J-1;
                               if I <= J then begin POM2    := POLE[I];
                                                    POLE[I] := POLE[J];
                                                    POLE[J] := POM2;
                                                    I := I+1;  J := J-1
                                              end
                        until I > J;
                        if I < D then begin S := S+1;
                                            Z[S].H := I;   Z[S].D := D;
                                      end;
                        D := J
                  until H >= D
            until S = 0
      end;                        { konec procedury QUICK }

   Procedure QUICKR(var POLE:A; N: INDEX);
     Procedure SORT(H,D : INDEX);
        var I, J : INDEX;  POM1, POM2 : SLOZKA;
        begin I := H;   J := D;
              POM1 := POLE[(H+D) div 2];
              repeat while POLE[I] < POM1 do I := I+1;
                     while POM1 < POLE[J] do J := J-1;
                     if I <= J then begin POM2    := POLE[I];
                                          POLE[I] := POLE[J];
                                          POLE[J] := POM2;
                                          I := I+1;  J := J-1
                                    end
              until I > J;
              if H < J then SORT(H,J);
              if I < D then SORT(I,D);
        end;                                      {konec procedury SORT}
     begin SORT(1,N)
     end;                                         {konec procedury QUICKR}
```

Použití kterékoliv z výše uvedených procedur k setřídění posloupnosti hodnot vyžaduje *stejné* definice a deklarace. Můžeme tedy vytvořit program např. ve tvaru:

```
Program SETRIDENI;
  const MAXPOCET= 30;
  type  SLOZKA  = integer;
        INDEX   = 0..MAXPOCET;
        A       = array[INDEX] of SLOZKA;
   var  POCET, K: INDEX;
        HODNOTY : A;
{*    Na tomto místě zapíšeme kteroukoliv z výše uvedených    *}
{*    deklarací procedur na řazení (např. proceduru SHUTT)    *}
  begin write('Zadej pocet nacitanych hodnot ke trideni: ');  readln(POCET);
        for K:=1 to POCET do begin write('Zadej ',K,'. hodnotu: ');
                                   readln(HODNOTY[K])
                             end;
{*    Na tomto místě zavoláme deklarovanou proceduru,         *}
{*    např. SHUTT(HODNOTY,POCET);                             *}
        writeln('Tisk setridenych hodnot:');
        for K:=1 to POCET do writeln(K,'. hodnota je ',HODNOTY[K]:4);
  end.
```

Metoda Shell sort, neboli Shellova[1] metoda řazení (řazení se snižujícím se přírůstkem) vychází z principu rozdělení řazených hodnot na 4 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o

[1]Honzík v [8] uvádí citaci: „Bohužel není snadné pochopit činnost této metody. Když se poprvé objevila v tisku, jistý

4 složky (1. skupinu tvoří 1., 5., 9. ... složka řazeného vektoru, 2. skupinu 2., 6., 10. ... atd). Každou skupinu seřadíme zvlášť nějakou známou metodou řazení. V další etapě rozdělíme pole řazených hodnot na 2 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o 2 složky (tj. složky 1., 3., 5. ... resp. 2., 4., 6. ...). V poslední fázi seřadíme celou posloupnost případnou výměnou sousedních dvojic.
Následující program má stejnou logiku jako poslední výše uvedený program. Za povšimnutí stojí pouze jiná definice typu A nutná pro použití procedury SHELL:

```
Program SETRIDENI;
  const MAXPOCET= 30;
        PS      = 4;
   type SLOZKA  = integer;
        INDEX   = 0..MAXPOCET;
        A       = array[-9..MAXPOCET] of SLOZKA;
   var  POCET, K: INDEX;
        HODNOTY : A;
   Procedure SHELL(var POLE:A; N: INDEX);
        type STRUKTURA = array[1..PS] of INDEX;
        var I, J, R, S : integer;
            POM        : SLOZKA;
            M          : 0..PS;
            PO         : STRUKTURA;
        begin PO[1]:=9;    PO[2]:=5;    PO[3]:=3;    PO[4]:=1;
              for M:=1 to PS do
                  begin R:=PO[M]; S:=-R;
                        for I:=R+1 to N do
                            begin POM:=POLE[I]; J:=I-R;
                                  if S=0 then S:=-R;
                                  S:=S+1;   POLE[S]:=POM;
                                  while POM<POLE[J] do
                                        begin POLE[J+R]:=POLE[J];
                                              J:=J-R
                                        end;
                                  POLE[J+R]:=POM
                            end
                  end
        end;                          { konec procedury SHELL }
   begin write('Zadej pocet nacitanych hodnot ke trideni: ');  readln(POCET);
         for K:=1 to POCET do begin write('Zadej ',K,'. hodnotu: ');
                                    readln(HODNOTY[K])
                              end;
         SHELL(HODNOTY,POCET);
         writeln('Tisk setridenych hodnot:');
         for K:=1 to POCET do writeln(K,'. hodnota je ',HODNOTY[K]:4);
   end.
```

K posouzení určité efektivnosti zmíněných metod poslouží následující tabulka časové náročnosti uvedených metod. Z tabulky je patrné, že na metodě bublinového třídění je zajímavý snad její název.

stav vektoru	setříděný		náhodný		opačně setř.	
počet prvků	256	512	256	512	256	512
Přímý výběr	489	1907	509	1956	695	2675
Přímé vkládání	12	23	366	1444	704	2836
Bubble sort	540	2165	1026	4054	1442	5931
Ripple sort	5	8	1104	4270	1645	6542
Shaker sort	5	9	961	3642	1619	6520
Shell sort	58	116	127	349	157	492
Quick sort	31	69	60	146	37	79

vedoucí programátor a systémový programátor, kteří nemohli pochopit její postup, vytvořili program a podrobili metodu řadě pokusů, v nichž dobře obstála. Přesto, že stále nechápali její tajuplný postup, zařadili ji do knihovny programů. ... Pokud se některý ze čtenářů domnívá, že by porozuměl Shellově metodě, může to zkusit“.

Závěrem tohoto příkladu ještě jednou poznamenejme, že tvar všech výše uvedených algoritmů zůstává zcela zachován i pro řazení číselných reálných hodnot, případně řetězců znaků, či jen znaků. V části definicí programu je třeba pouze změnit definici typu SLOZKA na např.

SLOZKA=real nebo SLOZKA=**string**[20] nebo SLOZKA=char atp.

konec Příkladu 4.1.3 konec Příkladu 4.1.3

Příklad 4.1.4: Matice celočíselných hodnot má rozměr M x N. (M<=20, N<=20). Na vstupu jsou hodnoty připraveny po řádcích, před prvním řádkem matice jsou hodnoty M a N. Zjistěte maximální hodnotu ze všech minimálních hodnot jednotlivých řádků. (Pro zjištění minima z řádku deklarujte podprogram).

Prvním úkolem, který je třeba vyřešit, je načtení hodnot matice. V našem případě je matice na vstupu v obecném tvaru:

$$\begin{array}{cccc} M & N & & \\ a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & & & \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{array}$$

Z uvedeného zadání nevyplývá ani pokyn ani nutnost řešit problém čtení matice v podobě podprogramu. Uvědomíme–li si však, že můžeme poměrně často řešit problematiku nějakého zpracování matice, pak se pravděpodobně pro deklaraci podprogramu na čtení matice rozhodneme. Takto jednou napsaný podprogram pak případně využíváme v celé řadě programů. Doplníme–li algoritmus pro čtení matice (uvedený v 1. kapitole) příkazy pro uvedení dialogu uživatele s počítačem a opatříme ho příslušnou hlavičkou podprogramu a částí lokálních definicí a deklarací, pak dostáváme podprogram CTIMAT v podobě procedury např. ve tvaru:

```
Procedure CTIMAT(var X:MAT;PR,PS:byte);
    var I,J:byte;
    begin for I:=1 to PR do
          begin writeln('Zadávej hodnoty ',I,'. řádku');
                for J:=1 to PS do
                begin write(' ':16,J,'. sloupce: ');
                      readln(X[I,J])
                end
          end
    end;
```

Řešením problému hledání minimální hodnoty z řady čísel jsme se již zabývali ve 2. kapitole. Nyní ze zadání vyplývá požadavek na deklaraci podprogramu řešícího uvedený problém. Vstupními hodnotami podprogramu MIN bude řada maximálně dvaceti (konkrétně (viz zadání) N) hodnot, které tvoří hodnoty jednoho řádku matice, struktury v podobě jednorozměrného pole (jednorozměrné pole–vektor). Výstupní hodnotou je 1 hodnota jednoduchého datového typu. Je tedy možné deklarovat požadovaný podprogram v podobě funkce, která může být např. ve tvaru:

```
Function MIN(Y:RADEK):integer;
    var I:byte;  MH:integer;
    begin MH:=maxint;
          for I:=1 to N do                { N je globální proměnná }
              if Y[I]<MH then MH:=Y[I];
          MIN:=MH
    end;
```

Příkazová část výsledného programu bude obsahovat zřejmě příkazy, které povedou k vyřešení těchto problémů:

- Načti hodnoty matice U. Tento příkaz vznikne kompozicí např. příkazů:

  write('Zadej rozměry vstupní matice:');
  readln(M,N);
  CTIMAT(U,M,N);

- Zpracuj hodnoty matice, tj. pro její jednotlivé řádky dělej:
  - vezmi hodnoty i–tého řádku
  - urči z nich hodnotu minimální
  - zjištěnou minimální hodnotu zpracuj takovým způsobem, aby ze všech takto zjištěných minimálních hodnot bylo možno určit hodnotu maximální

```
MAX:=-maxint;
for I:=1 to M do
begin for J:=1 to N do R[J]:=U[I,J];
      if MIN(R)>MAX then MAX:=MIN(R);
end;
```

nebo rychlejší:

```
MAX:=-maxint;
for I:=1 to M do
begin for J:=1 to N do R[J]:=U[I,J];
          POM:=MIN(R);
          if POM>MAX then MAX:=POM
end;
```

Výše uvedený algoritmus je možno také zapsat:

```
MAX:= -maxint;
for I:=1 to M do
    if MIN(U[I])>MAX then MAX:=MIN(U[I]);
```

nebo rychlejší:

```
MAX:=-maxint;
for I:=1 to M do
begin POM:=MIN(U[I]);
      if POM>MAX then MAX:=POM
end;
```

- Vytiskni požadované výsledky, tj. writeln('Hledaná hodnota je ',MAX);

Pak program řešeného problému může být případně ve tvaru:

```
Program MATICE;
   const MPRS   = 20;
   type  INDEX  = 1..MPRS;
         RADEK  = array[INDEX] of integer;
         MAT    = array[INDEX] of RADEK;
   var   M, N, I: INDEX;
         U      : MAT;
         MAX,POM: integer;
   Procedure CTIMAT(var X:MAT;PR,PS:INDEX);
        var I,J:INDEX;
      begin for I:=1 to PR do
            begin writeln('Zadávej hodnoty ',I,'. řádku');
                  for J:=1 to PS do
                  begin write(' ':16,J,'. sloupce: ');
                        readln(X[I,J])
                  end
            end
      end;
   Function MIN(Y:RADEK):integer;
       var I:INDEX;  MH:integer;
     begin MH:=maxint;
           for I:=1 to N do
               if Y[I]<MH then MH:=Y[I];
           MIN:=MH
     end;
begin write('Zadej rozmery matice MxN: '); readln(M,N);
      CTIMAT(U,M,N);
      MAX:=-maxint;
      for I:=1 to M do
      begin POM:=MIN(U[I]);
            if POM>MAX then MAX:=POM;
```

```
      end;
      writeln('Požadovaná hodnota je ',MAX)
end.
```

Modifikujeme-li zadání v tom smyslu, že máme například zjistit maximální hodnotu ze všech součtů jednotlivých řádků, pak celý program zůstává nezměněn kromě tvaru deklarace funkce MIN. Deklaraci funkce MIN zaměníme za deklaraci funkce např. SUMA ve tvaru:

```
   Function SUMA(Y:RADEK):integer;
      var   I:INDEX;  POMS:integer;
      begin POMS:=0;
            for I:=1 to N do POMS:=POMS+Y[I];
            SUMA:=POMS
      end;
```

Dále ještě nahradíme v příkazové části programu identifikátor funkce MIN identifikátorem funkce SUMA a další problém je vyřešen.

Na tomto místě ještě podotkněme, že identifikátor U je identifikátor *celé* matice, zápis U[I] identifikuje I-tý *řádek* matice a zápis U[I,J] identifikuje *prvek* matice o souřadnicích I-tý řádek a J-tý sloupec. Potřebujeme-li pracovat se sloupcem jako celkem, není přípustný zápis U[,J] a zápis U[J] identifikuje (viz výše) opět řádek matice, v tomto případě J-tý řádek. Je třeba obsah proměnné např. S definovat přiřazením:

J:=3; {potřebujeme J-tý sloupec (např. třetí)}
for I:=1 **to** M **do** S[I] := U[I,J];

a je-li proměnná S typu RADEK (je to možné nejen při alokaci paměťového místa pro čtvercovou matici, ale i pro matici s menším počtem řádků jak sloupců), pak může být skutečným parametrem jak pro funkci MIN, tak pro funkci SUMA. Tato skutečnost umožňuje modifikovat zadání řešeného příkladu i na hledání maxima ze součtů hodnot jednotlivých sloupců, či hledání maxima ze všech minimálních hodnot jednotlivých sloupců.

konec Příkladu 4.1.4 konec Příkladu 4.1.4

Příklad 4.1.5: Nechť existuje typ CLOVEK = (MUZ,ZENA) a platí:

+	MUZ	ZENA
MUZ	MUZ	ZENA
ZENA	ZENA	ZENA

∗	MUZ	ZENA
MUZ	MUZ	MUZ
ZENA	MUZ	ZENA

Vytvořte program, na základě kterého se přečtou ze vstupního standardního textového souboru dvě matice se složkami typu CLOVEK, vytisknou se, vypočte se součin obou matic a výsledná matice se vytiskne. Operace + a ∗ jsou definovány dle výše uvedených tabulek.

Řešení přináší dále uvedený program. Výčtovou hodnotu MUZ na vstupu reprezentuje čitelná hodnota 1, hodnotu ZENA pak rovněž celočíselná hodnota 0. Na výstupu je pak výčtová hodnota MUZ reprezentována pěticí znaků (řetězcem) ' MUZ ', hodnota ZENA pak řetězcem (hodnotou) ' ZENA'.

```
program LIDE;
   label 10;
   const PR=20; PS=20;                {* maximální rozměr matice 20x20 *}
   type CLOVEK=(MUZ,ZENA);
        MAT=array[1..PR,1..PS] of CLOVEK;
   var I,J,K,M,N1,N2,P :integer;
       SOUCIN,SOUCET : CLOVEK;
       ROD1,ROD2,ROD : MAT;
   procedure CTI(var MA:MAT; R,S:integer);{*      pro čtení matice      *}
       var Z:integer;                     {*      lokální proměnná      *}
       begin
         writeln('Jsem připraven pro vstup hodnot 1 (muž) nebo 0(žena) matice');
         for I:=1 to R do
             begin writeln('Zadávej postupně hodnoty ',I,'.řádku: ');
                   for J:=1 to S do begin read(Z);
```

```
                                        if Z=1 then MA[I,J]:=MUZ
                                               else MA[I,J]:=ZENA
                                        end;
                  readln;
             end;
                      {* neboť z INPUTu nelze číst hodnotu typu CLOVEK *}
   end;
procedure PIS(var MA:MAT; R,S:integer);   {*      pro tisk matice       *}
   begin for I:=1 to R do
             begin for J:=1 to S do if MA[I,J]=MUZ then write(' MUZ ')
                                                   else write(' ZENA');
                  writeln
             end
   end;
procedure SCIT(E,F:CLOVEK; var X:CLOVEK); {* pro sečítání dvou hodnot   *}
                                        {* typu CLOVEK dle definice   *}
                                        {* operace + ; E,F vstupní    *}
                                        {* X               výstupní   *}
                                        {* parametr procedury SCIT    *}

   begin if E=MUZ then if E=F then X:=MUZ
                               else X:=ZENA
                 else X:=ZENA
   end;
procedure NASOB(C,D:CLOVEK;var Y:CLOVEK); {* pro násobení dvou hodnot   *}
                                        {* typu CLOVEK dle definice   *}
                                        {* operace *                  *}
   begin if C=ZENA then if C=D then Y:=ZENA
                                else Y:=MUZ
                 else Y:=MUZ
   end;
begin write('Zadej rozměry obou matic (čtyři čísla):');
      readln(M,N1,N2,P);                {* čtení rozměrů obou matic   *}
                                        {* prvá matice rozměr M x N1  *}
                                        {* druhá matice rozměr N2 x P *}
      if N1<>N2 then begin writeln('MATICE NELZE NASOBIT');
                           goto 10
                     end;
        {* ukázka použití příkazu skoku. Příkaz skoku není nutno použít *}
        {* neboť lze pokračovat v části else úplného podm. příkazu      *}
      CTI(ROD1,M,N1); CTI(ROD2,N2,P);
      writeln(' MATICE A:'); writeln;
      PIS(ROD1,M,N1);
      writeln;
      writeln(' MATICE B:'); writeln;
      PIS(ROD2,N2,P);
      writeln;
                            {* začíná násobení matice ROD1 maticí ROD2 *}
      for I:=1 to M do
          for K:=1 to P do
            begin     {* prvý elementární součin inicializuje ROD[I,K] *}
                  NASOB(ROD1[I,1],ROD2[1,K],ROD[I,K]);
                  for J:=2 to N1 do
                    begin NASOB(ROD1[I,J],ROD2[J,K],SOUCIN);
                          SCIT(ROD[I,K],SOUCIN,SOUCET);
                          ROD[I,K]:=SOUCET
                    end
            end;
      writeln(' VYSLEDNA MATICE C:'); writeln;
```

```
      PIS(ROD,M,P);
      writeln;
10: end.

    Za předpokladu, že vstupní hodnoty máme ve tvaru: 4 3 3 6
                                                       1 0 0
                                                       0 1 0
                                                       0 0 1
                                                       1 1 1
                                                       1 1 1 0 0 0
je tvar výstupní sestavy:                              0 0 0 1 1 1
                        MATICE A:                      1 1 1 1 1 1
                        MUZ  ZENA ZENA
                        ZENA MUZ  ZENA
                        ZENA ZENA MUZ
                        MUZ  MUZ  MUZ

                        MATICE B:
                        MUZ  MUZ  MUZ  ZENA ZENA ZENA
                        ZENA ZENA ZENA MUZ  MUZ  MUZ
                        MUZ  MUZ  MUZ  MUZ  MUZ  MUZ

                        VYSLEDNA MATICE C:
                        ZENA ZENA ZENA MUZ  MUZ  MUZ
                        MUZ  MUZ  MUZ  ZENA ZENA ZENA
                        ZENA ZENA ZENA ZENA ZENA ZENA
                        MUZ  MUZ  MUZ  MUZ  MUZ  MUZ
```

Podotkněme ještě, že po záměně definice typu

CLOVEK = (MUZ,ZENA) za definici typu CLOVEK = (ZENA,MUZ)

nebo po změně dohody, že 1 reprezentuje na vstupu výčtovou hodnotu MUZ a 0 hodnotu ZENA, naopak, tj. že 1 reprezentuje hodnotu ZENA a 0 reprezentuje hodnotu MUZ,
by bylo v řadě implementací PASCALů možno v proceduře CTI nahradit příkaz

```
if Z=1 then MA[I,J]:=MUZ
       else MA[I,J]:=ZENA;
```

příkazem `MA[I,J] := CLOVEK(Z)`

konec Příkladu 4.1.5 konec Příkladu 4.1.5

4.2 Algoritmy nad záznamem

Příklad 4.2.1.: Deklarujte soubor LIDE typu CLOVEK a proměnnou OSOBA typu CLOVEK. Typ CLOVEK definujte jako záznam obsahující složky JMENO, PRIJMENI (řetězce znaků), ROKNAR (celé číslo), VZDELANI (výčet (zakladni, vyucen, stredni, vysoke)), POHLAVI (Boolean).

Řešení:

```
type CLOVEK = record JMENO,PRIJMENI : string[10];
                     ROKNAR         : integer;
                     VZDELANI       : (zakladni,vyucen,stredni,vysoke);
                     POHLAVI        : Boolean
              end;
var LIDE    : file of CLOVEK;
    OSOBA   : CLOVEK;
```

Rozšíříme–li zadání tak, že u záznamu CLOVEK místo složky ROKNAR požadujeme složku NAROZEN, která nám má poskytovat údaje o dnu, měsíci a roku narození, pak je třeba složku ROKNAR typu integer nahradit složkou ROKNAR typu NAROZEN, přičemž typ NAROZEN můžeme definovat jako záznam buď

```
NAROZEN : record DEN, MESIC, ROK : integer
          end;
```

popřípadě mnoha jinými způsoby, např.

```
NAROZEN : record DEN   : 1..31;
                 MESIC : (leden,unor,brezen,duben,kveten,cerven,
                          cervenec,srpen,zari,rijen,listopad,prosinec);
                 ROK   : 1890..1999
          end;
```

Rozšíříme–li dále zadání o možnost sledovat

- u žen
  - POCDETI (celé číslo z intervalu např. 0..8)
  - STAV (výčet např. (svobodna,vdana,rozvedena,vdova))
  - příjmení za svobodna DIVCI (řetězec znaků)
- u mužů
  - BRANNYPOMER (výčet např.(nevojak,vojak), případně Boolean)

pak řešením za předpokladu, že obsahem složky POHLAVI je hodnota *false* mají být obsahem celé proměnné OSOBA (typu záznam) informace o ženě, a za předpokladu, že obsahem složky POHLAVI je hodnota *true* budou v záznamu informace o muži. Proměnnou OSOBA je tedy třeba deklarovat jako proměnnou typu *záznam s variantními složkami*. Řešení požadovaných definicí a deklarací již s ohledem na zdůvodňované doporučení o vhodnosti provádět deklaraci proměnné buď pomocí identifikátoru standardního typu, nebo identifikátoru typu definovaného v odstavci **type** (a ne popisem typu tak, jak jsme to u tohoto příkladu dosud dělali) může mít tvar:

```
type  RET    = string[10];
      INT1   = 1..31;
      INT2   = 1890..2000;
      INT3   = 0..10;
      VYC1   = (leden,unor,brezen,duben,duben,kveten,cerven,cervenec,
                srpen,zari,rijen,listopad,prosinec);
      VYC2   = (zakladni, vyucen,stredni,vysoke);
      VYC3   = (svobodna,vdana,rozvedena,vdova);
      DATUM  = record DEN   : INT1;
                      MESIC : VYC1;
                      ROK   : INT2;
               end;
      CLOVEK = record JMENO,PRIJMENI : RET;
                      NAROZEN        : DATUM;
                      VZDELAN        : VYC2;
                      case POHLAVI   : Boolean of
                         false : (POCDETI     : INT3;
                                  DIVCI       : RET;
                                  STAV        : VYC3);
                         true  : (BRANNYPOMER : Boolean)
               end;
      SOUBOR = file of CLOVEK;
var   LIDE   : SOUBOR;
      OSOBA  : CLOVEK;
```

Variantní část záznamu je možno psát také jinak, např. takto:

```
record .
       .
       case POHLAVI : (zena,muz) of
         ZENA : (POCDETI     : INT3;
                 DIVCI       : RET;
                 STAV        : VYC3);
         MUZ  : (BRANNYPOMER : (nevojak,vojak))
end;
```

konec Příkladu 4.2.1 konec Příkladu 4.2.1

Poněvadž praktické využití záznamů je především při práci s netextovými soubory (se soubory typu záznam), jsou další příklady s algoritmizací nad záznamy uvedeny v následujícím odstavci (Příklad 4.3.2 (jeho poslední uvedené řešení) a Příklad 4.3.5).

4.3 Algoritmy nad souborem

Způsob práce se soubory je značně závislý na vlastní implementaci jazyka. Normy uvedené ve standardním PASCALu jsou v každé implementaci více či méně porušovány a součástí téměř každé další implementace jsou nové podprogramy pracující nad soubory, které buď v jiné implementaci nenacházíme, popř. jejich použití je jiné.[2]

4.3.1 Práce s textovými soubory

Student, který má již za sebou první zkušenosti s laděním programu pracujícího nad proměnnou typu pole, může snad mít smíšené pocity z hlediska přínosu počítačového zpracování problému. Po kratší či delší době se mu podařilo odladit formální (syntaktické) chyby a konečně došlo ke spuštění jeho programu. Zadal vstupní data (např. vstupní data z Příkladu 4.1.5) a ejhle: počítač sice počítá, avšak očekávané výsledky se neukazují. Takováto situace může nastat z mnoha důvodů. V Příkladě 4.1.5 dojde k přerušení programu při načítání vstupních hodnot např. tehdy, když se spleteme a místo 0 (nuly) klepneme na klávesnici O (písmeno O). K neočekávaným, tj. špatným, výsledkům může docházet např. tehdy, když v některém ze tří cyklů potřebných k vynásobení matice ROD1 a ROD (uvedeno na přelomu stran 55 a 56) uvedeme špatnou koncovou hodnotu cyklu (proměnnou, jejímž obsahem je hodnota větší jak 20, nebo je koncová hodnota jiná, než má být, tj. např. místo P jsme napsali M). Logickou chybu je třeba najít a odstranit a program překladu spustíme znovu. Opět zadáváme vstupní hodnoty. Není jich mnoho, ale když je zadáváme popáté, podesáté a jen pomalu se dostáváme ke kýženému výsledku (již máme např. pouze „rozházenou“ výstupní sestavu), jsme již pěkně otráveni. A přesto je řešení jednoduché. Vstupní hodnoty napíšeme do textového souboru (nějakým editorem stejně, jak jsme psali program), který pod názvem např. DATA.TXT budeme mít třeba na disketě zastrčené do mechaniky A (nebo na pevném disku v našem adresáři, nebo kdekoliv jinde). S těmito vstupními daty budeme pracovat tak dlouho, dokud program neodladíme – vstupy budeme číst nikoliv z klávesnice, ale ze souboru DATA.TXT.

Nadeklarujeme–li v programu uvedeném v Příkladě 4.1.5 proměnnou

F : text;

místo prvého příkazu

readln(M,N1,N2,P)

uvedeme příkazy

[2]Tak např. porovnejme implementaci Turbo–Pascalu a HP–Pascalu v oblasti otevírání souborů:

	otevření fyzického souboru pro čtení	otevření fyzického souboru pro zápis	otevření fyzického netextového souboru pro modifikaci
Turbo–Pascal	assign(F,JMENO); reset(F)	assign(F,JMENO); rewrite(F)	assign(F,JMENO); reset(F)
HP–Pascal	reset(F,JMENO)	rewrite(F,JMENO)	open(F,JMENO)
Standard	Nezbytná je hlavička programu obsahující mimo jiné proměnné typu soubor, které při spouštění programu jsou nahrazeny názvy konkrétních souborů		
	reset(F)	rewrite(F)	

kde JMENO je název fyzického souboru (řetězec znaků)
a F je identifikátor proměnné typu soubor (logický soubor)

```
                                assign(F,'A:DATA.TXT'); reset(F);
                                readln(F,M,N1,N2,P)
```

a proceduru CTI nahradíme procedurou CTI ve tvaru

```
Procedure CTI(var MA:MAT;R,S:integer);
    var Z:integer;
    begin for I:=1 to R do
              for J:=1 to S do begin read(F,Z);
                                     if Z=1 then MA[I,J]:=MUZ
                                            else MA[I,J]:=ZENA
                                end;
end;
```

pak program nečte hodnoty z klávesnice, ale z jiného textového souboru, souboru DATA.TXT.

Příklad 4.3.1: Na vstupu jsou informace o zaměstnancích podniku. O každém ze zaměstnanců je k dispozici jeho
- příjmení (řetězec znaků)
- měsíční příjem (celé kladné číslo)
- rok narození (celé číslo z intervalu 1900..2000)
- údaj o pohlaví (např. 0 reprezentuje ženu, 1 muže)

Úkol: Zjistěte průměrný měsíční příjem v podniku.

Příklad tohoto typu by nás již neměl překvapit. Je obdobou příkladu z odstavce 1.3 pouze s tím rozdílem,
že místo příkazů

```
write('Zadej plat:');          readln(PLAT);
```

můžeme psát příkazy

```
write('Zadej příjmení:');      readln(PRIJ);
write('Zadej plat:');          readln(PLAT);
write('Zadej rok narození:');  readln(NAROZ);
write('Zadej pohlaví:');       readln(SEX);
```

a jednoduchou úpravou algoritmu můžeme zjistit průměrný měsíční příjem žen v podniku (totéž u mužů), průměrný měsíční příjem zaměstnanců starších než 30 a mladších než 40 roků apod.

Vyzkoušíme-li např. program

```
Program  ZAMESTNANCI;
   type  RETEZ = string[16];          INT1        = 1900..2000;
         INT2  = 0..1;
   var   PRIJ  : RETEZ;               NAROZ       : INT1;
         PLAT  : word;                SEX         : INT2;
         POCET : byte;                SUMA, PRUMER: real;
   begin POCET:=0, SUMA:=0.0;
         write('Zadej příjmení:');  readln(PRIJ);
         While PRIJ <> ''  do
         begin write('Zadej plat:');  readln(PLAT);
               write('Zadej rok narození:');readln(NAROZ);
               write('Zadej pohlaví (0 žena,1 muž):'); readln(SEX);
               if SEX=0 then begin SUMA:=SUMA+PLAT;
                                   POCET:=POCET + 1
                             end;
               write('Zadej příjmení:');readln(PRIJ);
         end;
         PRUMER = SUMA/POCET;
         writeln('Požadovaná hodnota je ', PRUMER:8:2)
    end.
```

pak zjistíme průměrný plat ženy v podniku. Nahradíme-li podmínku

if SEX = 0

podmínkou

if (SEX=1) **and** (NAROZ<1963) **and** (NAROZ>1953)

pak obdržíme průměrný plat muže staršího třiceti a mladšího čtyřiceti let (vzhledem k roku 1993) a obdobně můžeme formulovat mnoho dalších podmínek, přičemž vždy zůstává zbytek programu nezměněn.

Máme-li připravena vstupní data (zaměstnanců v podniku může být třeba 17, ale též 4273) např.

```
Kousalová     3840  1958  0
Doupal        4400  1943  1
⋮

Krejčí        3650  1963  0
```

můžeme program spustit, zadávat vstupy, on nám dá výsledek, my program upravíme nahrazením podmínky jinou podmínkou, opět jej spustíme, zadáváme vstupy, dostáváme výsledek a tak dále. I při poměrně malém počtu zadávaných zaměstnanců nás brzy omrzí bušit do klávesnice stále stejné informace. A jsme u jádra věci. Napišme připravená vstupní data do souboru na disketu. Jak? No přece stejně, jak jsme napsali program: třeba v editoru Turbo-Pascalu. A již můžeme začit kolem sebe šířit, jak skvělý program máme, a jistě se najde mnoho zájemců, kteří budou chtít jeho služeb využívat. Není problém. I oni si mohou vytvořit *svůj soubor svých zaměstnanců* na disketu. Ale pozor! Všechny soubory musí být formálně stejné - přizpůsobené programu (ten při vzniku souborů ještě ani nemusí existovat, pak naopak program přizpůsobíme požadavkům na strukturu souborů). Pořizovači souborů dostanou např. následující příkaz:

- jeden řádek souboru bude obsahovat informace právě o jednom zaměstnanci
- na řádku budou informace umístěny takto:
  - prvých šestnáct míst bude nést informaci o příjmení zaměstnance (program počítá pro tuto informaci s proměnnou typu **string** [16])
  - dále budou číselné hodnoty (tři), které budou vzájemně odděleny číselným oddělovačem (tj. alespoň jednou mezerou) v tomto pořadí:
    - plat (celé číslo),
    - rok narození (číslo z intervalu 1900..2000) a
    - údaj o pohlaví (0 – půjde-li o ženu, 1 – půjde-li o muže).
  - první ze tří číselných hodnot může tedy začínat nejdříve na 17. sloupci

Až se k nám budou dostávat postupně diskety s pořízenými vstupními soubory, jistě již budeme mít upravený program, který bude schopen tyto soubory akceptovat. Program může mít tvar např.:

```
Program  ZAMEST1;
   type  RETEZ = string[16];          INT1        = 1900..2000;
         INT2  = 0..1;
   var   PRIJ,JMSOUB : RETEZ;         NAROZ       : INT1;
         PLAT        : word;          SEX         : INT2;
         POCET       : byte;          SUMA, PRUMER: real;
         F           : text;
   begin POCET:=0;     SUMA:=0;
         writeln('Vlož svoji disketu se souborem do mechaniky A');
         write('a napis jméno souboru:');  readln(JMSOUB);
         assign(F,JMSOUB); reset(F);
         while not eof(F) do
         begin readln(F,PRIJ,PLAT,NAROZ,SEX);
               if SEX=0 then begin SUMA:=SUMA+PLAT;
                                   POCET:=POCET + 1
                             end;
         end;
         close(F);
         PRUMER:= SUMA/POCET;
         writeln('Požadovaná hodnota je ', PRUMER:8:2)
    end.
```

Vidíme tedy, že programy ZAMEST1 a ZAMESTNANCI jsou si velice podobny, přičemž ZAMEST1 je nesporně „profesionálnější“, i když do opravdové profesionality mu chybí ještě mnoho. Je jistě žádoucí obecnější řešení podmínky, kdy uživatel v dialogu s počítačem odpovídá, za jakých podmínek se má hodnota PLAT zpracovávat (v závislosti na pohlaví, v závislosti na věku, v závislosti na kombinaci obojího apod.). Rovněž je třeba v praktických aplikacích předcházet možným haváriím programu v důsledku chybné činnosti uživatele, popř. haváriím v důsledku v programu neošetřených, ale přesto reálně nastalých situací (plat je záporný $\Longrightarrow$ chybné výsledky). V těchto souvislostech se jedná např. i o ošetření možné neexistence souboru v aktuálním adresáři aktivního disku.

Každá implementace PASCALu má řadu standardních podprogramů pro přímý styk s operačním systémem. Často používanou funkcí při práci se soubory je v Turbo-Pascalu funkce IOResult. Ta při správně provedené vstupní/výstupní operaci (tou je nejen operace čtení/zápisu, ale i jiné operace nad souborem, např. jeho otevření) vrací hodnotu 0. Nadeklarujeme-li funkci

```
Function JESOUBOR(JMENO:RETEZ) : Boolean;
   var   P : text;      POM : Boolean;
   begin assign(P,JMENO);
         {$I-}
             reset(P);
         {$I+}
         POM := IOResult=0;
         if POM then close(P);
         JESOUBOR := POM;
   end;
```

pak ji po zařazení do části definicí a deklarací výše uvedeného programu můžeme v jeho příkazové části použít. Začátek příkazové časti může být např. ve tvaru:

```
   begin POCET:=0, SUMA:=0;
         writeln('Vlož svoji disketu se souborem do mechaniky A');
         write('a napiš jméno souboru:');  readln(JMSOUB);
         while not JESOUBOR(JMSOUB) do
             begin writeln('Zadal jsi špatné jméno (takový soubor neexistuje)');
                   writeln('Vlož svoji disketu se souborem do mechaniky A');
                   write('a napiš znovu jméno souboru:');  readln(JMSOUB);
             end;
         writeln('Již se zadařilo');
         assign(F,JMSOUB); reset(F);
         .
         .
```

Na tomto místě opět upozorňujeme studenty s vyššími ambicemi v oblasti programování na řadu potřebných podprogramů v knihovnách SYSTEM a DOS Turbo–Pascalu (např. v [3]), jakož i na problematiku direktiv překladače (např. v doplněném vydání [10]).

konec Příkladu 4.3.1 konec Příkladu 4.3.1

Příklad 4.3.2: Na základě textového souboru ZAM.TXT, jehož struktura je popsána v Příkladě 4.3.1, vytiskněte abecedně setříděný seznam všech žen s platem nad 4500,– Kč.

Setřídění textového souboru in situ (na místě) není záležitostí jednoduchou a i pro časovou náročnost V/V operací nad vnějšími soubory se nedoporučuje provádět. Vhodné je potřebné informace z textového souboru nejprve vybrat (načíst) do vhodné proměnné v operační paměti, poté provést seřazení, např. některou z metod uvedených v Příkladě 4.1.3 a nakonec požadované informace vytisknout.

Zásadní otázkou je použití „vhodné“ proměnné operační paměti. Jaká to musí být proměnná? Zřejmě taková, aby se do ní vešly všechny požadované informace. Požadujeme–li seznamy žen obsahující jejich příjmení, plat a rok narození, pak v okamžiku řazení musíme mít tyto údaje v operační paměti. Vhodnou proměnnou tedy jistě bude proměnná typu pole. Pole příjmení, pole platů a pole roků narození těch zaměstnanců, kteří nás zajímají — žen s platem nad 4500 Kč.

Za předpokladu definicí a deklarací ve tvaru

```
const   MPZ          =  30;
 type   RETEZ        =  string[16];
        INT1         =  1900..2000;
        INT2         =  0..1;
        POLE1        =  array[1..MPZ] of RETEZ;
        POLE2        =  array[1..MPZ] of word;
        POLE3        =  array[1..MPZ] of INT1;
  var   PRIJ,POM1    :  RETEZ;
        NAROZ,POM3   :  INT1;
        PLAT,POM2    :  word;
        I,J          :  byte;
        SEX          :  INT2;
        PRIJMENI     :  POLE1;
        PLATY        :  POLE2;
        DATA         :  POLE3;
```

pak příkazy pro definování obsahu příslušných složek polí budou mít tvar:

```
assign(F,'ZAM.TXT'); reset(F);
I:=1;
while not eof(F) do
    begin readln(F,PRIJ,PLAT,NAROZ,SEX);
          if (SEX = 0) and(PLAT > 4500) then
             begin PRIJMENI[I]:=PRIJ;
                   PLATY[I]:=PLAT;
                   DATA[I]:=NAROZ;
                   I:=I + 1
             end;
    end;
close(F);
```

Takto vzniknou datové struktury potřebných příjmení, platů a roků narození. Struktury je třeba setřídit podle příjmení. Použijeme–li např. algoritmus uvedený v Příkladu 4.1.3 jako Shuttle sort, pak píšeme:

```
J:=1;
while J<=(I-2) do                          { I−1 je počet všech zajímavých žen }
       if PRIJMENI[J] > PRIJMENI[J+1]
          then begin POM1:=PRIMENI[J];
                     PRIJMENI[J]:=PRIJMENI[J+1];
                     PRIJMENI[J+1]:=POM1;
                     POM2:=PLATY[J]; PLATY[J]:=PLATY[J+1];
                     PLATY[J+1]:=POM2;
                     POM3:=DATA[J];
                     DATA[J]:=DATA[J+1]; DATA[J+1]:=POM3;
                     J:=1;
               end
          else J:=J+1;
```

Všimněme si, že proměnná POM1 je stejného typu jako *složka* pole příjmení, proměnná POM2 je stejného typu jako *složka* pole PLATY a POM3 je stejného typu jako *složka* pole DATA.

Čtenář by měl v této fázi opět vycítit „manuální náročnost“ algoritmu řazení seznamu. Vždyť prakticky třikrát opisoval tytéž tři příkazy pro výměnu dvou hodnot. Je třeba přijít s myšlenkou. Proč přehazovat složky tří struktur? Nešlo by to vyřešit se strukturou jednou? Ale jistě, použijeme–li strukturu v podobě pole, jehož složkou bude nehomogenní struktura, tj. struktura typu záznam. Za předpokladu

následujících definicí (s využitím výše uvedených definicí)

```
type  SLOZKA  =  record  PR     :  RETEZ;
                         MZDA   :  word;
                         RN     :  INT1
                 end;
      POLE    =  array   [1..MPZ]  of  SLOZKA
```

je proměnná ZENY deklarovaná jako

```
var  ZENY  :  POLE;
     POM   :  SLOZKA
```

právě tou datovou strukturou, kterou potřebujeme. Pak program založený na strategii výběru potřebných údajů do pole záznamů, řazení nad tímto polem a tisk setříděného seznamu, může mít např. tvar:

```
Program SEZNAM;
   const MPZ    = 30;
   type  RETEZ  = string[16];     INT1 = 1900..2000;     INT2 = 0..1;
         SLOZKA = record PR   : RETEZ;
                         MZDA : word;
                         RN   : INT1
                   end;
         POLE   = array[1..MPZ] of SLOZKA;
   var   PRIJ   : RETEZ;          NAROZ  : INT1;
         PLAT   : word;           SEX    : INT2;
         F      : text;           I, J   : byte;
         ZENY   : POLE;           POM    : SLOZKA;
   begin assign(F,'B:ZAM.TXT'); reset(F);
         I := 0;
         while not eof(F) do
            begin readln(F,PRIJ,PLAT,NAROZ,SEX);
                  if (SEX=0) and (PLAT>4500) then begin I := I + 1;
                                                        ZENY[I].PR   := PRIJ;
                                                        ZENY[I].MZDA := PLAT;
                                                        ZENY[I].RN   := NAROZ;
                                                   end;
            end;
         close(F);
         J := 1;
         while J <= (I-1) do                  { I je počet všech zajímavých žen}
            if ZENY[J].PR > ZENY[J+1].PR then begin POM       := ZENY[J];
                                                    ZENY[J]   := ZENY[J+1];
                                                    ZENY[J+1] := POM;
                                                    J := 1
                                              end
                                         else J := J + 1;
         writeln('Abecedni seznam hledanych zen:');
         for J:=1 to (I) do
             with ZENY[J] do writeln(PR,MZDA:6,RN:6)
   end.
```

konec Příkladu 4.3.2 konec Příkladu 4.3.2

Příklad 4.3.3: Napište program, který ve vybraném textovém souboru vyhledá a vytiskne všechna slova zvolené délky. Konkrétní soubor i požadovanou délku slov si volí uživatel z klávesnice.

Uvědomíme-li si, že textový soubor S je tvořen z jednotlivých řádků a řádek je tvořen z elementů — jednotlivých znaků, a právě pouze jeden znak nás v určitém okamžiku zajímá, pak dospějeme k nutnosti

použít dvojitý cyklus charakteristický právě pro typ příkladů, jejichž jádrem je zpracování informace v podobě jednoho znaku:

```
while not eof(S) do
      begin while not eoln(S) do
            begin read(S,ZNAK);
                  .
                  .           {*  zpracování načteného znaku    *}
                  .
            end;
            readln(S);        {*  odhození "vyčteného" řádku    *}
      end;
```

Nyní již postačuje uvědomit si, co vše může být slovním oddělovačem a můžeme napsat program např. ve tvaru:

```
Program SLOVA;
   const ODDEL = [' ', '.', ',', ';','?','!','(',')','[',']','{','}',':','=',''''];
   type  RET         = string[63];
   var   DOPIS       : text;         DELKA, RADEK, POCET : integer;
         PRVNI, ZNAK : char;         JM, SLOVO           : RET;
   begin write('Kde mám vzít vstupní text?  '); readln(JM);
         assign(DOPIS,JM);  reset(DOPIS);
         writeln('Našel jsem dopis');
         write('Zadej požadovanou délku slova: '); readln(DELKA);
         SLOVO := '';
         while not eof(DOPIS) do
         begin while not eoln(DOPIS) do
               begin read(DOPIS,ZNAK);
                     while not(ZNAK in ODDEL) and not eoln(DOPIS) do
                     begin SLOVO := SLOVO + ZNAK;
                           read(DOPIS,ZNAK);
                     end;
                     if not (ZNAK in ODDEL) then SLOVO:=SLOVO+ZNAK;
                     if length(SLOVO) = DELKA then writeln(SLOVO);
                     SLOVO:= '';
               end;
               readln(DOPIS)
         end;
        close(DOPIS);
   end.
```

Hodláme-li tisknout všechna slova začínající písmenem (rozlišujeme mezi velkým a malým písmenem) zadávaným uživatelem z klávesnice, pak příkazová část algoritmu může mít tvar:

```
begin write('Kde mám vzít vstupní text?  '); readln(JM);
      assign(DOPIS,JM); reset(DOPIS);
      writeln('Zadej první písmeno: '); readln(PRVNI);
      SLOVO := '';
      while not eof(DOPIS) do
      begin while not eoln(DOPIS) do
            begin read(DOPIS,ZNAK);
                  while not(ZNAK in ODDEL) and not eoln(DOPIS) do
                  begin SLOVO := SLOVO+ZNAK;
                        read(DOPIS,ZNAK);
                  end;
                  if not (znak in ODDEL) then SLOVO:=SLOVO+ZNAK;
                  if (length(SLOVO)>0) and (SLOVO[1]=PRVNI) then write(SLOVO,'   ');
                  SLOVO:= '';
            end;
```

```
            readln(DOPIS)
      end;
      close(DOPIS);
end.
```

Diskutabilní u obou algoritmů je, zda např. K231 je či není z našeho hlediska slovo. Oba algoritmy uvedenou posloupnost čtyř znaků za slovo považují. Pakliže se domníváme, že slovním oddělovačem je *jakýkoliv* znak mimo písmene (malého či velkého), pak je možno opustit strategii vytváření slova

dokud není znakem oddělovač, tj. **if not** ZNAK **in** ODDEL **then** ...

a nahradit ji strategií vytváření slova ze znaků

dokud je znak písmeno, tj. **if** ZNAK **in** ['A'..'Z','a'..'z'] **then** ...

Dalším pěkným problémem je ošetření možnosti existence rozděleného slova, jehož vyřešení necháváme na čtenáři.

konec Příkladu 4.3.3 konec Příkladu 4.3.3

Příklad 4.3.4: Napište program, který z informací umístěných v souboru OLD.TXT vytvoří soubor NEW.TXT obsahující požadované (výběrové) informace.

Za předpokladu, že v původním (kmenovém) souboru je spousta informací, pak tvorba „podsouboru" je poměrně častá činnost při uspokojování nejrůznějších požadavků na poskytnutí informací výběrových. Máme-li v původním souboru o studentech mj. i informace o jméně, příjmení a jednotlivých známkách získaných studentem v semestru, pak můžeme vytvořit soubor, který bude obsahovat pouze dvě informace: příjmení studenta a jeho průměrnou známku. Jednoduché řešení přináší následující program:

```
Program NOVYSOUBOR;
  type  RET1           = string[10];
        RET2           = string[15];
  var   STARY, NOVY    : text;        Z1, Z2, Z3, Z4 : integer;
        JMENO          : RET1;        PRIJMENI       : RET2;
  begin assign(NOVY,'a:new.txt');     rewrite(NOVY);
        assign(STARY,'a:old.txt');    reset(STARY);
        while not eof(STARY) do begin readln(STARY,JMENO,PRIJMENI,Z1,Z2,Z3,Z4);
                                      writeln(NOVY,PRIJMENI,(Z1+Z2+Z3+Z4)/4:6:2);
                                end;
        close(NOVY);
  end.
```

konec Příkladu 4.3.4 konec Příkladu 4.3.4

4.3.2 Práce s netextovými soubory

Zatímco v textovém souboru se lépe orientuje člověk, jsou netextové soubory (soubory ve vnitřní reprezentaci) efektivnější pro práci počítače. Již víme, co se děje v okamžiku, kdy počítač plní příkaz „Načti celočíselnou hodnotu z textového souboru (např. z klávesnice) do operační paměti" (viz Příklad 3.5).

Vytvořme (např. pro řadu následných aplikací) z textového souboru celočíselných hodnot soubor typu integer programem

```
Program PREVOD;                    { * převod čísel zadávaných z klávesnice *}
                                   { * do netextového souboru              *}
    var   S : file of integer;   X : integer;
    begin assign(S,'CISLA.TYP'); rewrite(S);
          write('Zadej číselnou hodnotu (ukončení pomocí -999):');readln(X);
          while X<> -999 do begin write(S,X);
                                  write('Zadej další hodnotu:');
                                  readln(X)
                            end;
          close(S);
    end.
```

nebo programem

```
Program PREVOD2;                    { * převod čísel z textového souboru * }
                                    { * do souboru netextového           * }
    var SVSTUP  : text;
        SVYSTUP : file of integer;    X       : integer;
    begin assign(SVSTUP,'CISLA.TXT'); reset(SVSTUP);
          assign(SVYSTUP,'CISLA.TYP'); rewrite(SVYSTUP);
          while not eof(SVSTUP) do begin read(SVSTUP,X);
                                         write(SVYSTUP,X);
                                   end;
          close(SVYSTUP)
    end.
```

Netextový soubor má spoustu výhod. Zpravidla zabírá méně místa než textový soubor se stejnými informacemi. Byl–li textový soubor CISLA.TXT ve tvaru např.

$$\begin{array}{rrrrr} -437 & 836 & -12411 & 653 & 286 \\ 4393 & 838 & -30442 & 666 & 381 \\ -138 & 453 & & & \end{array}$$

pak zabíral minimálně 58 B (počet v něm umístěných znaků včetně značek konců řádků). Soubor CISLA.TYP obsahuje rovněž 12 celočíselných hodnot, na jejichž uložení spotřebuje 24 B (zabírá-li jedna hodnota typu integer 2 B).

Příklad 4.3.5: Vytvořte netextový soubor ZAM.TYP s využitím proměnné LIDE typu soubor tak,

jak jsme o ní hovořili v Příkladě 4.2.1

Netextový soubor člověk vytváří na základě souboru textového. Tato zdánlivě zbytečná činnost je motivována jednak snahou o efektivní využití paměťového místa (viz příklad výše), jednak snahou o možnost psát přehledné (tudíž i modifikovatelné a prodejné) programy. Jak již víme, lze z textového souboru číst hodnoty typu znak (char), hodnoty typu celé číslo (integer), hodnoty typu desetinné číslo (real) a často i řetězce znaků (**string** resp. **packed array of** char).

Budeme mít tedy určité problémy s načtením hodnot složek POHLAVI (typu Boolean) a STAV (typ definovaný výčtem) popř. BRANNYPOMER (typ Boolean). Je zapotřebí zvolit určité reprezentanty pro ty hodnoty, které jsou z textového souboru nečitelné.

Reprezentanty jednotlivých identifikátorů uvedených ve výčtu volíme zpravidla ve tvaru celých čísel od 0 (pro první identifikátor) až po $N-1$ (pro N-tý identifikátor). V tom případě můžeme vedle standardní funkce `ord` provádějící převod hodnoty ordinálního typu na hodnotu typu integer (ordinální, pořadové číslo) používat nestandardní, avšak ve většině implementací Pascalů realizované možnosti převodu hodnoty ordinálního typu na hodnotu jiného ordinálního typu se stejným ordinálním číslem.

Pak řešení úlohy je za předpokladu platných definicí a deklarací v Příkladě 4.2.1 v podobě

```
var OSOBA      : CLOVEK;   REPREZ     : 0..3;    POM1       : 0..1;
.
.
assign(LIDE,'ZAM.TYP'); rewrite(LIDE);
assign(VSTUP,'ZAM.TXT'); reset(VSTUP);
while not eof(VSTUP) do
      begin with OSOBA, NAROZEN do
            begin read(VSTUP,JMENO,PRIJMENI,DEN,MESIC,ROK,REPREZ,POM1);
                  VZDELANI := VYC2(REPREZ);
                  POHLAVI  := POM1=1;              {* popř. POHLAVI:= Boolean(POM1) *}
                  if POHLAVI then begin readln(VSTUP,POM1);
                                        BRANNYPOMER:=Boolean(POM2);
                                                   {* popř. BRANNYPOMER:=POM2=1     *}
                                  end
                             else
                                  begin readln(VSTUP,POCDETI,DIVCI,REPREZ);
                                        STAV:=VYC3(REPREZ)
```

```
                                end;
            end;
            write(LIDE,OSOBA)
      end;
close(LIDE);
.
.
```

Podotýkáme ještě, že vstupní textový soubor může mít např. tvar:

```
Jana    Nováčková  24  6 1955  0  2  MLADA
Jan     Nováček    12 10 1938  1  1
.
.
Honza   Jakoubek   01 07 1964  1  1
```

Odměnou za práci vloženou do tranformace textového souboru na netextový soubor nám je radost z přehlednosti i stručnosti programů pracujících nad netextovým souborem. Tisk středoškolaček se třemi dětmi pořídíme na základě algoritmu

```
.
.
while not eof(LIDE) do begin read(OSOBA);
                             with OSOBA do
                                  if POHLAVI=zena then
                                     if VZDELANI=stredni then
                                        if POCDETI=3 then writeln(JMENO,PRIJMENI)
                       end;
.
.
```

konec Příkladu 4.3.5 konec Příkladu 4.3.5

Příklad 4.3.6: Problematika řazení souboru

I když řazení informací přímo v souboru tímto způsobem nedoporučujeme, přesto si jeho řešení dovolíme čtenáři předložit. Nechceme v něm hned v jeho počátečních krocích algoritmizací vzbudit dojem, že by něco „ na počítači nešlo“.

```
Program TRIDSOUB;
type JMENO  = string[10];
var PR,A,B  : JMENO;    S       : file of JMENO;
    F       : text;     I,J,P   : byte;
begin assign(F,'SOUB.TXT');   reset(F);
      assign(S,'SOUB.TYP');   rewrite(S);
      writeln('Opis textového souboru:');
      while not eof(F) do begin readln(F,PR);
                                writeln(PR);    write(S,PR);
                          end;
      { je vytvořen netextový soubor S z původního textového souboru F}
      close(F); close(S);
      reset(S); P:=0;
      writeln('Opis typového souboru:');
      while not eof (S) do begin read(S,PR);
                                  writeln(PR);
                                  P:=P+1;
                           end;
      close(S);
      { Textový soubor nelze řadit, lze řadit soubor netextový (typový)}
      {Poněvadž takový soubor jsem vytvořil - soubor S - řadím}
```

```
      {Obyčejná bublinová metoda}
      for I:=1 to P-1 do
      begin reset(S);
            for J:=1 to P-1 do begin read(S,A); read(S,B);
                                     if A>B then begin seek(S,filepos(S)-2);
                                                       write(S,B);
                                                       write(S,A);
                                                 end;
                                     seek(S,filepos(S)-1);
                               end;
            close(S);
      end;
      { Procedura seek vlastně posouvá v souboru přístupovou}
      { (tj. čtecí nebo zápisovou hlavu na místo, se kterým chceme "cloumat"}
      reset(S);
      writeln('Opis setříděného souboru:');
      while not eof(S) do begin read(S,PR);
                                writeln(PR);
                          end;
      close(S);
end.
```

konec Příkladu 4.3.6 konec Příkladu 4.3.6

4.4 Algoritmy nad množinou

Příklad 4.4.1: Napište program pro své spoluobčany, jenž jim sdělí, zda obec vašeho trvalého bydliště

má přímé autobusové spojení s okresním městem JM kraje, jež je zadáváno z klávesnice. Při řešení pracujte s typem množina.

Řešení definicí a deklarací, jakož i inicializace proměnné SPOJENI typu množina je celkem logické:

```
type OKRESY = (Blansko,Brno,Breclav,Hodonin,Jihlava,Kromeriz,
               Prostejov, Trebic,Uh.Hradiste, Vyskov,Zlin,Znojmo,Zdar);
     MNSPOJ = set of OKRESY;
var  SPOJENI : MNSPOJ;
begin SPOJENI := [Blansko,Jihlava,Zlin,Znojmo];
        ⋮
```

Typ OKRESY by mohl být dán výčtem všech okresů ČR (je jich méně než 256) a proměnná SPOJENI nabývá hodnoty odvozené od skutečnosti, vztahující se k obci trvalého bydliště uživatelů.

Zadá-li uživatel MISTO, do kterého chce cestovat, pak požadovaná odpověď je získána příkazem

```
if MISTO in SPOJENI then writeln('Existuje spojení')
                    else writeln('Neexistuje spojení');
```

Problémem je skutečnost, že proměnná MISTO je proměnnou výčtového typu (**var** MISTO:OKRESY), a tudíž do ní nelze načíst hodnotu přímo z klávesnice. Je třeba tady zajistit převod „čitelné“ reprezentace vstupující informace na potřebnou informaci typu výčet (OKRESY). Čitelnou reprezentací může být textový kód okresu v podobě dvou znaků tak, jak jej známe z poznávacích značek automobilů (SPZ). Program můžeme zapsat ve tvaru:

```
Program AUTOBUSY;
   type OKRESY = (Blansko,Brno,Breclav,Hodonin,Jihlava,Kromeriz,Prostejov,
                  Trebic,Uh_Hradiste,Vyskov,Zlin,Znojmo,Zdar,Nic);
        MNSPOJ = set of OKRESY;
   var  SPOJENI: MNSPOJ;
        MISTO  : OKRESY;
```

```
      Z1, Z2 : char;
begin SPOJENI := [Blansko, Jihlava, Zlin, Znojmo];
      repeat writeln('Zadej okresní město, do kterého hledáš spojení');
             write('                        (dvě velká pismena SPZ): ');
             readln(Z1,Z2);
             MISTO := Nic;
             case Z1 of
                'B': if (Z2='M') or (Z2='O') then MISTO := Brno;
                'H': if Z2='O' then MISTO := Hodonin;
                'J': if Z2='I' then MISTO := Jihlava;
                'K': if Z2='R' then MISTO := Kromeriz;
                'P': if Z2='V' then MISTO := Prostejov;
                'T': if Z2='R' then MISTO := Trebic;
                'U': if Z2='H' then MISTO := Uh_Hradiste;
                'V': if Z2='Y' then MISTO := Vyskov;
                'Z': case Z2 of
                        'L': MISTO := Zlin;
                        'N': MISTO := Znojmo;
                        'R': MISTO := Zdar;
                     end;
             end;
             if MISTO in SPOJENI then writeln(' ':50,'Existuje spojení')
                                 else writeln(' ':50,'Neexistuje spojení');
             writeln;
             write('Požaduješ hledat další spojení  A/N : '); readln(Z1);
      until not(Z1 in ['A','a']);
end.
```

konec Příkladu 4.4.1 konec Příkladu 4.4.1

Příklad 4.4.2: Vytvořte program, který bude schopen uživateli sdělit, zda se svými šesti čísly vyhrál

první, druhou, třetí nebo čtvrtou cenu, či zda nevyhrál.

Předpokládejme, že existuje 6 tažených čísel (TAH) a 6 vsazených čísel (VSADIL). Jaké paměťové místo bude obsazovat proměnná TAH a proměnná VSADIL? No zřejmě tak velké, aby se do něj vlezlo po šesti hodnotách z intervalu např. 1..49. Proměnné TAH a VSADIL mohou být proměnnými typu pole:

```
type  CISLA       : 1..49;
      POLE        = array[1..6] of CISLA;
 var  TAH,VSADIL  : POLE;
```

Pak řešení našeho problému tkví ve zjištění, kolik hodnot z pole VSADIL je v poli TAH obsaženo:

```
POCET:=0;
for I:=1 to 6 do
    for J:=1 to 6 do if TAH[I] = VSADIL[J] then POCET:=POCET + 1;
```

Podmínka TAH[I] = VSADIL[J] se vyhodnocuje celkem 36×.

Proměnné TAH a VSADIL nemusí být typu pole, ale mohou být i jiného strukturovaného datového typu. Deklarujeme proměnnou TAH typu množina

```
type  CISLA  = 1..49;
      POLE   = array[1..6] of CISLA;
      MNOZ   = set of CISLA;
 var  TAH    : MNOZ;
      CISLA  : POLE;
```

Pak řešení přináší příkazy

```
  POCET:=0;
for J:=1 to 6 do if VSADIL[J] in TAH then POCET:=POCET + 1;
```

Požadovaný program pak může mít tvar:

```
Program  SPORTKA;
    type CISLA= 1..49;
         POLE = array[1..6] of CISLA;
         MNOZ = set of CISLA;
    var  TAH  : MNOZ;
         CISLA: POLE;
         X    : CISLA;        POCET,I;byte;       Z :char;
    begin writeln('Zadej tažená čísla:'); TAH:=[];
          for I:=1 to 6 do
              begin write(I,'.tažené číslo:'); readln(X);
                    TAH:=TAH + [X];
              end;
          repeat writeln('Zadej vsazená čísla:');
                 for I:=1 to 6 do
                     begin write(I,'.vsazené číslo:'); readln(CISLA[I]);
                     end;
                 POCET:=0;
                 for I:=1 to 6 do if CISLA[I] in TAH
                                     then POCET:=POCET + 1;
                 case POCET of
                    6: writeln('Blahopřeji - 1.cena');
                    5: writeln('Blahopřeji - 2.cena');
                    4: writeln('Blahopřeji - 3.cena');
                    3: writeln('Blahopřeji - 4.cena');
                  2,1,0: writeln('Bohužel - snad příště');
                 end;
                 write('Máš další vsazená čísla? - A/N:');
                 readln(Z);
          until not(Z in['A','a']);
    end.
```

Poznamenejme ještě, že příkaz **case** (obsáhlý, ale přehledný), můžeme nahradit příkazem **if** (kratším a s myšlenkou) ve tvaru:

if POCET $<$ 3 **then** writeln('Bohužel – snad příště')
else writeln('Blahopřeji – ',(7 – POCET),'.cena');

Jistě by bylo také možné deklarovat

var TAH,CISLA : MNOZ;

a použít příkazy
POCET:=0;
for I:=1 **to** 49 **do if** I **in** (TAH*CISLA) **then** POCET:=POCET+1;
modifikované na rychlejší
POCET:=0; POM:= TAH*CISLA;
for I:=1 **to** 49 **do if** I **in** POM **then** POCET:=POCET+1;
Podmínka I **in** POM se vyhodnocuje 49x. Proměnná POM je typu množina MNOZ a její obsah je definován průnikem množin TAH a CISLA.

konec Příkladu 4.4.2 konec Příkladu 4.4.2