7. ŘAZENÍ

Třída algoritmů nazývané tradičně "třídění" byla již mnohokrát zpracována mnoha významnými autory a výsledky jejich teoretických prací i rozsáhlých experimentů byly publikovány v nepřeberném množství monografií, statí, článků a příspěvků. Snad proto se bude zdát poněkud neortodoxní skutečnost, že tato kapitola významně čerpá z diplomové práce vynikajícího bulharského studenta Čavdara Pačeva [1], vedené autorem skript. Hlavním důvodem je to, že v práci jsou nejzávažnější výsledky známých autorů plně implementovány na běžně dostupném počítači (EC 1033) v jazyce Pascal (překladač Stony-Brook). Pro jednotlivé algoritmy je v této práci původní metodou odvozena časová a prostorová složitost, provedeny rozsáhlé experimenty a jejich výsledky úspěšně konfrontovány s výsledky publikovanými především Wirthem [3]. Nechť je tedy tato kapitola nejen zdrojem jinde nepublikovaných výsledků, ale také projevem uznání dobré práci našich studentů i stimulem pro ty, které taková práce ještě očekává.

7.1. Základní pojmy

Význam pojmů "třídění", "uspořádání", "řazení" a "setřídění" se na základě běžných zkušeností zdá být velmi podobný, ne-li identický. Definujme neformálně sémantiku těchto pojmů (abstraktních operací nad lineární datovou strukturou) pro potřeby disciplíny programování tak, jak je definována v návrhu revidované názvoslovné normy ČSN [5].

Třídění (angl. sorting, sort) je rozdělování údajů na skupiny údajů se stejnými vlastnostmi.

Uspořádání podle klíčů (angl. collating) je seřazení údajů podle prvků (klíčů) lineárně uspořádané množiny.

Řazení (angl. sequencing) je uspořádání údajů podle relace lineárního uspořádání.

Slučování (angl. coalescing) je vytváření souboru sjednocením několika souborů.

Setřídění (také zakládání; angl. merging) je vytváření souboru sjednocením několika souborů, jejichž údaje jsou seřazeny podle téže relace uspořádání se zachováním této relace.

Ve smyslu těchto definic budeme algoritmy tradičně nazývané "třídicí algoritmy" nazývat "řadicí algoritmy" a řazení budeme chápat jako zvláštní případ obecnějšího pojmu třídění. V anglickém označení jednotlivých algoritmů však nadále budeme používat jazykový kmen "sort" ve smyslu řazení tak, jak se používá v anglické i ruské terminologii.*)

*) Termín "třídění" vznikl v předpočítačové éře, kdy se mechanizované řazení údajů na děrných štítcích provádělo postupným tříděním štítků na třídicích strojích. Tyto stroje roztřídily (rozdělily) soubor štítků na 10 podsouborů podle hodnoty 0-9 vyděrované v daném sloupci. Operátor obsluhující stroj seřadil ručně tyto podsoubory za sebe a mohl dát třídit soubor podle jiného sloupce. Třídil-li se takto soubor postupně např. podle sloupců 10,9,8,7,6 byl nakonec seřazen podle velikosti celých čísel vyděrovaných ve sloupcích 6-10. Tento princip je zachován v metodě "řazení podle základu" (Radix-sort). V pozdější době byly na počítačích používány algoritmy řazení, které vůbec třídění nevyužívaly, ale název spojený s tříděním jim v široké obci uživatelů již zůstal... Protože však přesné myšlení se může odrážet jen v přesné terminologii, nebudeme se tradice názvu třídění přidržovat. Ostatně ani v tělocviku nevelíme "Setřiďte se podle velikosti" ....

V souvislosti s vlastnostmi řazení zavedeme některé další pojmy :

Sekvenčnost řazení je vlastnost, která vyjadřuje, že řadicí algoritmus pracuje se vstupními údaji i s datovými meziprodukty v tom pořadí v jakém jsou lineárně uspořádány v datové struktuře. Opakem sekvenčnosti je přímý přístup k jednotlivým položkám vstupní nebo pomocné struktury. O algoritmu, který nepracuje sekvenčně říkáme, že je nesekvenční.

Prostorová a časová složitost označuje míru prostoru resp. času potřebnou k realizaci daného algoritmu. U řadicích algoritmů ji budeme vyjadřovat jako funkci počtu n řazených prvků. Chování složitosti pro $n \to \infty$ se nazývá asymptotickou složitostí. Pro označení asymptotického růstu složitostních funkcí budeme používat Knuthovu notaci, která popisuje vztah mezi dvěmi funkcemi f(n) a g(n) pomocí vyhodnocování podílu f/g pro dostatečně velká n.

Budeme říkat, že funkce f roste maximálně tak rychle jako funkce g, a zapisovat to

$$f(n) = O(g(n)),$$ *)

když existují konstanty $n_0 \geq 1$ a $c > 0$ takové, že pro všechna $n \geq n_0$ platí $f(n) \leq c * g(n)$ a dále budeme říkat, že funkce f roste minimálně tak rychle jako funkce g, a zapisovat to

$$f(n) = \Omega(g(n)),$$

když existují konstanty $n_0 \geq 1$ a $c > 0$ takové, že pro všechna $n \geq h_0$ platí $f(n) \geq c * g(n)$.

Budeme říkat, že funkce f roste řádově stejně rychle jako g, a zapisovat to

$$f(n) = \Theta(g(n))$$

když platí současně $f(n)=O(g(n))$ a $f(n) = \Omega(g(n))$

Přirozenost řazení je vlastnost algoritmu, která vyjadřuje, že doba potřebná k řazení již seřazené množiny údajů je menší než doba pro seřazení náhodně uspořádané množiny a ta je menší než doba pro seřazení opačně seřazené množiny údajů.

Stabilita řazení je vlastnost řazení, která vyjadřuje, že algoritmus zachová vzájemné pořadí údajů se shodnými klíči. Stabilita je nutná tam, kde se vyžaduje, aby se při řazení údajů podle klíče s vyšší prioritou neporušilo pořadí údajů se shodnými klíči vyšší priority, získané předcházejícím řazením množiny podle klíčů s nižší prioritou (viz řazení podle více klíčů).

7.1.1. Řazení podle více klíčů

Předpokládejme, že máme vytvořit dva seznamy osob z daného souboru osob s využitím jejich data narození. V prvním seznamu budou osoby od nejstaršího k nejmladšímu; druhý seznam bude sloužit jako přehled o narozeninách. Osoby se stejným datem narozenin budou seřazeny od nejstaršího k nejmladšímu. Předpokládejme, že soubor neobsahuje dvě stejně staré osoby.

Nechť je definován typ

```
TYPDATNAR=record
            ROK:1800..2000;
            MESIC:1..12;
            DEN:1..31
            end
```

*) O(g(n)) je funkce velké Omikron

Pro první seznam mají klíče sestupnou prioritu v pořadí ROK,MESIC,DEN, zatímco pro druhý seznam mají sestupnou prioritu v pořadí MESIC,DEN,ROK.

K řešení vedou dva přístupy :

a) Vytvoření složené relace uspořádání, která může mít tvar funkce, ve které se budou postupně srovnávat klíče se snižující se prioritou. Dojde-li k nerovnosti u klíčů vyšší priority, je relace rozhodnuta. Jinak se postupuje k nižší prioritě. Uveďme příklad funkce PRVNISTARSI pro řazení prvního seznamu :

```
function PRVNISTARSI (PRV,DRUH:TYPDATNAR):Boolean;
begin
  if PRV.ROK > DRUH.ROK
     then PRVNISTARSI:=true
     else if PRV.ROK < DRUH.ROK
            then PRVNISTARSI:=false
            else if PRV.MESIC > DRUH.MESIC      { Roky jsou stejné }
                   then PRVNISTARSI:=true
                   else if PRV.MESIC < DRUH.MESIC
                          then PRVNISTARSI:=false
                          else if PRV.DEN < DRUH.DEN    { Roky i měsíce jsou stejné
                                 then PRVNISTARSI:=true
                                 else PRVNISTARSI:=false
end
```

Pomocí takové relace lze seřadit osoby podle stáří v jedním řazením , Čtenář nechť si odvodí podobnou funkci pro seřazení seznamu narozenin.

b) Postupné řazení podle jednotlivých klíčů dosáhne téhož seřazení, bude-li se postupovat v řazení od nižší priority klíče k vyšší prioritě. T.zn., že seznam osob podle stáří získáme, budeme-li daný soubor postupně řadit podle klíčů DEN, MESIC, ROK a seznam narozenin získáme postupným řazením podle klíčů MESIC,DEN,ROK . Podmínkou pro úspěšné seřazení je stabilita řadicí metody, která např. pro první seznam zaručí, že se zachová vzájemné pořadí osob narozených ve stejném roce. Toto pořadí je výsledkem předchozích kol řazení, které zaručily, že dříve byla ta osoba, která má v roce narozeniny dříve. Princip postupného řazení podle více klíčů je využit v metodě řazení podle základu (Radix-sort).

7.1.2. Řazení bez přesunu položek

Nejvýznamnějšími operacemi každého algoritmu řazení jsou porovnání dvou položek a přesun položky (např. výměna dvou položek). Počet těchto operací, potřebný k realizaci seřazení určuje časovou složitost řadicí metody. Zabírá-li položka, která se má přesouvat, větší paměťový prostor, pak její přesun je časově náročný. Je-li celá seřazovaná lineární struktura ve vnitřní paměti, lze algoritmus řazení významně zrychlit tak, že se nebudou přesouvat položky, ale pouze jejich ukazatele v pomocném poli.

Nechť jsou dány typy :

```
TYPPOLOZKY=record KLIC:TYPKLIC;
                  DATA:TYPDATA
               end;
TYPPOLE=array [1..N] of TYPPOLOZKY
```

pak lze vytvořit pomocné pole ukazatelů (indexů)

```
TYPPOMPOLE=array [1..N] of 1..N;
```

Na počátku se POMPOLE inicializuje tak, aby každý prvek měl hodnotu svého indexu

```
for I:=1 to N do POMPOLE[I]:=I;
```

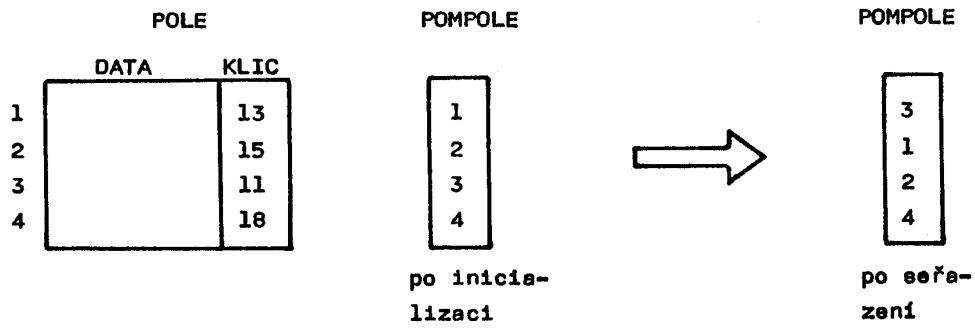
Relace mezi prvky I_1 a I_2 bude mít v algoritmu řazení tvar např. :

```
POLE [POMPOLE[I1]].KLIC < POLE[POMPOLE[I2]].KLIC
```

a odpovídající přesun (výměna) bude mít tvar :

```
POM:=POMPOLE[I1];        {var POM:1..N}
POMPOLE[I1]:=POMPOLE[I2];
POMPOLE[I2]:=POM
```

Výsledkem je, že pole POMPOLE obsahuje indexy položek seřazovaného pole, seřazení podle velikosti klíčů položek. Situaci znázorňuje obr. 7.1.



Obr. 7.1. Řazení bez přesunu položek

Po seřazení lze do výstupního pole v jednom průchodu zapsat seřazené pole cyklem :

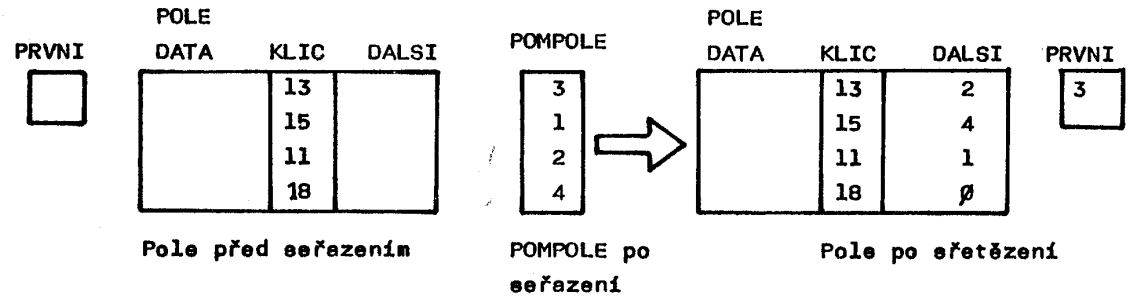
```
for I:=1 to N do VYSTPOLE[I]:=POLE[POMPOLE[I]] ;
```

Jiná možnost je sřetězení řazeného pole tak, aby se vytvořila posloupnost položek seřazených podle velikosti klíčů. Situaci odvozenou z obr. 7.1. znázorňuje obr. 7.2.

Zřetězení provede úsek programu :

```
PRVNI:=POMPOLE[1];
for I:=1 to N-1 do POLE[POMPOLE[I]].DALSI:=POMPOLE[I+1];
POLE[POMPOLE[N]].DALSI:=Ø;  { Nula ve funkci nilu }
```

Seřazenou posloupnost, která je zřetězena v poli, můžeme bez pomocného pole sekvenčně seřadit podle vtipného algoritmu, který vytvořil Mac Laren a který je převzat po úpravě z [3].



Obr. 7.2. Řazení bez přesunu se zřetězením

Nejnižší prvek seřazené posloupnosti je dán proměnnou PRVNI. V cyklu algoritmu sepostupně prochází polem s krokem 1 pomocí indexu i zřetězeným seznamem, pomocí ukazatele POM. Prvek na indexu i a POM se vzájemně vymění (v algoritmu je pro záměnu použit operátor :=:) a do ukazatele i-tého prvku se uloží jeho původní poloha a do POM se uloží následník. Byl-li ale následník přesunut výměnou dozadu, nalezne se odkazem pro ukazatel v i-tém prvku.

Celý algoritmus, který není snadné napoprvé vždy pochopit, má tvar :

```
i:=1; POM:=PRVNI;
while i < n do
     begin
          while POM < i do POM:=POLE[POM].DALSI;
                { Hledání následníka přesunutého na pozici větší než i }
          POLE[i]:=:POLE[POM]; { výměna prvků }
          POLE[i].DALSI:=:POM; { výměna ukazatelů }
          i:=i+1   { Prvních i-1 prvků již je na svém místě }
     end;
```

7.1.3. Rozdělení algoritmů řazení

Klasifikaci řadicích algoritmů můžeme provádět podle různých kriterií. Zmíníme se stručně o nejzávažnějších z nich.

a) Dělení podle typu paměti v níž je řazená struktura uložena

Metody vnitřního (interního) řazení se nazývají také metody řazení polí. Předpokládají uložení seřazované struktury v operační paměti a přímý (nesekvenční) přístup k položkám struktury.
Metody vnějšího (externího) řazení se nazývají také metody řazení souborů. Předpokládají sekvenční přístup k položkám seřazované struktury. Zvláštní skupinu mohou tvořit struktury s indexsekvenčním přístupem (implementované na magnetických discích, bubnech), u nichž metody řazení často kombinují principy vnitřního i vnějšího řazení.

b) Dělení podle typu procesoru

Na seriovém (sekvenčním) procesoru implementujeme seriové (sekvenční)algoritmy řazení. Myslíme tím, že následující operace algoritmu se může zahájit až po dokončení předcházející. Na paralelním procesoru lze implementovat paralelní algoritmy. Paralelní procesor umožňuje současný průběh více operací. (Všimněme si, že v tělocvičně se řadíme podle paralelního algoritmu, a že je

nesporně rychlejší, než kdybychom se řadili sekvenčně)

c) Dělení na přímé a nepřímé metody

Přímé metody využívají daného principu řazení v nejjednodušší podobě. Jsou jednoduché, ale mají vysokou časovou složitost (obvykle $O(n^2)$). Jsou vhodné pro malý počet seřazovaných položek.
Nepřímé metody vycházejí z týchž principů jako metody přímé, ale vylepšují základní princip různými programovacími metodami. Jsou složitější pro zápis algoritmu i pro jeho pochopení, ale mají nižší časovou složitost (často $O(n \log_2 n)$). Jsou vhodné pro větší počet seřazovaných položek.

d) Dělení podle základního principu řazení

- Metody pracující na principu výběru (angl. selection) přesouvají postupně největší (nejmenší) prvek ze seřazované množiny do výstupní lineární struktury.
- Metody pracující na principu vkládání (angl. insertion) zařazují (zatřiďují) do seřazené výstupní lineární struktury postupně všechny prvky seřazované množiny (v libovolném pořadí).
- Metody pracující na principu rozdělování (angl. partition) rozdělují postupně všechny (pod)množiny na dvě další podmnožiny tak, že všechny prvky jedné podmnožiny jsou menší než všechny prvky druhé podmnožiny.
- Metody pracující na principu setřídění (sdružování; angl. merging) sjednocují seřazené podmnožiny do větších seřazených podmnožin.
- Metody pracující na jiných principech vytváří méně sourodou skupinu různých principů nebo kombinací základních principů.

Dá se říci, že principy výběru a vkládání jsou zvláštními případy principů rozdělení a setřídění, kdy jedna z množin sestává z jediného prvku.

Kapitola o řazení se bude zabývat především vnitřními seriovými metodami všech základních principů. V závěru kapitoly budou stručně uvedeny základní principy vnějších řadicích metod.

7.1.4. Zásady hodnocení časové a prostorové složitosti řadicích algoritmů

Časová a prostorová složitost algoritmů se určuje s pomocí vlastností modelů procesoru zpracovávajících analyzované algoritmy. Pro účely řazení se v publikovaných zdrojích většinou používá model procesoru, v němž se započítávají pouze operace přesunů a porovnání. Ostatní operace se zanedbávají.

U algoritmů uvedených v této kapitole budou uvedeny výsledky analýzy získané na základě původní metody uvedené v [1]. Základem analýzy je hypotetický procesor, který umožňuje odvozovat časovou i paměťovou složitost algoritmů přímo z programů v Pascalu a který respektuje základní vlastnosti počítače JSEP, na němž byly teoretické vztahy konfrontovány s experimentem. Základní i strukturované příkazy jsou oceněny odvozeným počtem časových i paměťových kvant. Jednoduché i strukturované údaje (s jistými omezeními) jsou oceněny odpovídajícím počtem paměťových kvant i počtem časových kvant potřebných k jejich zpřístupnění. Pro ocenění bylo zvoleno rovnoměrné váhové kriterium, které předpokládá, že pro provedení každé instrukce je zapotřebí jedna časová jednotka a že slovo zabírá jednu paměťovou jednotku. Protože se bere v úvahu i mechanismus volání procedury, jsou jednotlivé algoritmy prezentovány ve formě procedur bez parametrů, v nichž je řazené pole globálním objektem. Vztahy, které budou u jednotlivých metod uvedeny, jsou odvo-

zeny v [1] a platí pro pole sestávající z n položek, kde položku tvoří pouze klíč typu integer (=4 paměťové kvanta). Dále bude u jednotlivých metod uveden skutečný čas naměřený pro daný algoritmus na počítači EC 1033 (uvedený v 10^{-2}s) pro různé hodnoty n a pro seřazené pole (SP), náhodně uspořádané pole (NUP) a opačně seřazené pole (OSP).

7.1.5. Význam časové složitosti algoritmů

Podle některých údajů [4] se řazením spotřebuje až 15% strojového času počítačů. Zdá se, že tedy stojí zato, zabývat se soustavněji časovou složitostí právě u této třídy algoritmů. Ze zvyšující se rychlostí počítačů může plynout, že úsilí pro tvorbu rychlejších algoritmů má stále menší význam. Situace je však poněkud složitější. S rostoucí rychlostí lze řešit rozsáhlejší problémy. Předpokládejme, že k řešení daného problému je k dispozici 5 různých algoritmů. Tab. 7.1. ukazuje, jak rozsáhlé problémy lze řešit jednotlivými algoritmy, má-li řešení trvat 1 s, 1 minutu a 1 hod. [9].

Prvá část tab. 7.1. ukazuje, jak se zvětší rozsah řešitelného problému, zvýší-li se rychlost počítače 10x. Zatímco při složitosti (n) se rozsah zvýší 10x, při složitosti (n^2) již jen asi 3x. Vezmeme-li za základ 1 minutu resp. 1 hod., můžeme podle tab. 7.1. při náhradě algoritmu A_4 algoritmem A_3 řešit 6x resp. 12x větší problém; náhradou A_4 algoritmem A_2 dokonce 125x resp. 1307x větší problém. V tab. 7.1. je složitost uvedena jen řádově; význam však má i koeficient úměrnosti tohoto řádu. Nechť je časová složitost pro algoritmus A_1=1000n, pro A_2= = 100 n $\log_2 n$, pro A_3=10 n^2, pro A_4=n^3 a pro A_5=2^n. Pak algoritmus A_5 bude nejlepší pro $2 \leq n \leq 9$, A_3 pro $10 \leq n \leq 58$, A_2 pro $59 \leq n \leq 1024$ a A1 pro $n > 1024$!

Algoritmus	Složitost	Max. rozsah problému (n)			Vliv zrychlení 10x na rozsah	
		1 s	1 min	1 hod	rozsah před zrychlením	rozsah po zrychlení
A_1	n	1000	6.10^4	$3.6.10^6$	S_1	$10xS_1$
A_2	n $\log_2 n$	140	4893	2.10^5	S_2	$\approx 10xS_2$
A_3	n^2	31	244	1897	S_3	$3.16xS_3$
A_4	n^3	10	39	153	S_4	$2.15xS_4$
A_5	2^n	9	15	21	S_5	$S_5+3.3$

Tab. 7.1. Rozsah řešitelných problémů

7.1.6. Smluvené konvence pro řadicí algoritmy

Ve všech následujících algoritmech budeme předpokládat :

a) Řazené pole je deklarováno jako

```
var A : array [1..N] of integer;
```

Pro zjednodušení budeme tedy uvažovat pouze pole klíčů. V procedurách řazení bude toto pole globálním objektem.

b) Nebude-li definováno jinak, budeme pojmem seřazené pole rozumět pole seřazené podle velikosti klíčů vzestupně.

c) Pro zkrácení a zpřehlednění zápisu programu bude pro zápis vzájemné výměny hodnot dvou proměnných téhož typu použita notace

```
a :=: b
```

která je ekvivalentní zápisu

```
p:=a;
a:=b;
b:=p      {kde p je pomocná proměnná téhož typu jako a a b.}
```

d) Všechny funkce zapisované jako log jsou logaritmické funkce při základě 2 - tedy $\log_2$ ...

e) Všechny vztahy pro časovou a paměťovou složitost jsou odvozeny v [1] pro pole klíčů, kde klíč typu integer zabírá 4 byty

f) Všechny experimenty byly provedeny s procedurou uvedenou v textu kapitoly na počítači EC 1033. Uvedené časy jsou v 10^{-2} s.

7.2. Řazení na principu výběru

7.2.1. Metoda přímého výběru (Straight selection-sort)

Princip této metody je velmi prostý: předpokládejme, že část pole od indexu 1 až do K je seřazena a část pole od indexu K+1 do n je neseřazena. Minimální prvek z části pole od K+1 do n se vymění s (K+1)ním prvkem, a za seřazenou považujeme část pole od 1 do K+1. Na počátku je seřazená část pole prázdná a poslední hledání minima se provádí pro prvky (n-1) a n. Symetrickým způsobem (pro seřazení od nejmenšího k největšímu) je hledání maxima a jeho výměna s posledním prvkem neseřazené části pole. Seřazená část pole je "vzadu" a rozšiřuje se zprava doleva.

Princip algoritmu lze popsat v Pascalu takto :

```
for I:=1 to N-1 do
   begin  najdi minimální prvek pole mezi indexy i a n a
          index minima ulož do K;
          A[I]:=:A[K] {výměna prvků A[I] a A[K]}
   end
```

Procedura řazení pak bude mít tvar :

```
procedure SELECT_SORT;
var I,J,K : 1..n; POM:integer;
begin for I:= 1 to N-1 do
        begin K:=I; POM:=A[K]; {Inicializace hledání minima}
              for J:=I+1 to N do  {Hledání minima}
                  if A[J]< POM then begin POM:=A[J];
                                          K:=J
                                    end;
              A[I]:=:A[K] {výměna}
        end
end  {Procedury}
```

Výsledky analýzy této metody lze shrnout takto :

a) Metoda je nestabilní. Při výměně minima s i-tým prvkem se i-tý prvek může dostat za prvek se shodnou hodnotou klíče a tím se může porušit jejich relativní pořadí ve výsledném seřazeném poli.

b) Pro průměrnou časovou složitost platí vztah :

$$T_s(n) \approx 4.5\ n^2 + 11\ n + 4.17\ (n-1)\ \log_2\ (n-1) - 11.5$$

c) Pro maximální časovou složitost platí vztah :

$$T_{MAX}(n) \approx 6n^2 + 19.5\ n - 20$$

d) Pro paměťovou složitost platí vztah :

$$S(n) = 4n + 171$$

e) Experimentálně byly naměřeny tyto výsledky :

n	128	256	512
OSP	64	254	968
NUP	50	212	744

7.2.2. Metoda bublinového výběru (Bubble-sort)

Základní princip této metody je shodný s předcházející metodou. Odlišný a také časově náročnější je způsob vyhledávání minima (resp. maxima) a jeho záměna s prvním (resp. posledním prvkem). Neseřazená část pole se prochází zprava doleva (resp. zleva doprava) a porovnávají se postupně každé dva sousední prvky pole. Nejsou-li vzájemně seřazeny ve smyslu výsledného seřazení, vymění se. Tímto způsobem "vyplave" minimum (resp. maximum) na nejlevější (nejpravější) pozici neseřazené části pole. I ostatní prvky mohou změnit svou pozici. Např. maximální (resp. minimální prvek) se posune o 1 pozici směrem ke svému řádnému místu.

Uvedená metoda má řadu algoritmicky zajímavých variant. Žádná z nich však nepřináší kvalitativně lepší výsledky.

Tzv. Přirozená metoda bublinového výběru ukončí vnější cyklus tehdy, nedojde-li ve vnitřním cyklu k výměně žádné dvojice.

Varianta této metody (zvaná někdy Ripple-Sort) si pamatuje pozici první dvojice u které došlo k výměně. V příštím cyklu začíná porovnávat až od předcházející dvojice.

Varianta zvaná Shaker-Sort prochází pole střídavě zleva-doprava a zprava-doleva. Seřazené části pole jsou v průběhu řazení na obou koncích pole a při ukončení řazení se spojí. Myšlenky obou předchozích variant lze v této variantě uplatnit.

Varianta zvaná Shuttle-Sort pracuje tak, že dojde-li u dvojice k výměně (např. při průchodu zleva-doprava), vrací se metoda s prvkem, který se posunuje doleva tak dlouho, dokud dochází k výměně. Pak se vrací do pozice u níž ukončila posun doprava a pokračuje směrem vpravo. Metoda končí, porovná-li úspěšně poslední dvojici prvků.

Přirozená metoda bublinového výběru má tvar :

```
procedure BUBBLESORT;
var I,J:1..N;
    KONEC:Boolean;
begin I:=2;
      repeat
           KONEC:=true;
           for J:= N downto I do   {cyklus porovnávání dvojic}
              if A[J-1] > A [J]
                   then begin KONEC:=false;
                              A[J-1]:=:A[J]
                        end;
           I:=I+1
      until KONEC or (I=N+1)
end {procedury BUBBLESORT}
```

Z analýzy této metody vyplývají tyto závěry :

a) Metoda se chová stabilně a přirozeně. Je ze všech metod nejrychlejší, řadí-li se již seřazené pole. Její princip je vhodný pro testování, zda je pole seřazené. V jiných případech je nejméně účinnou metodou !

b) Pro maximální časovou složitost platí vztah :

$$T_{MAX}(n) \doteq 11{,}5\ n^2 + 25{,}5\ n - 36$$

c) Pro paměťovou složitost platí vztah :

$$S(n) = 4\ n + 175$$

d) Experimentálně byly naměřeny tyto hodnoty :

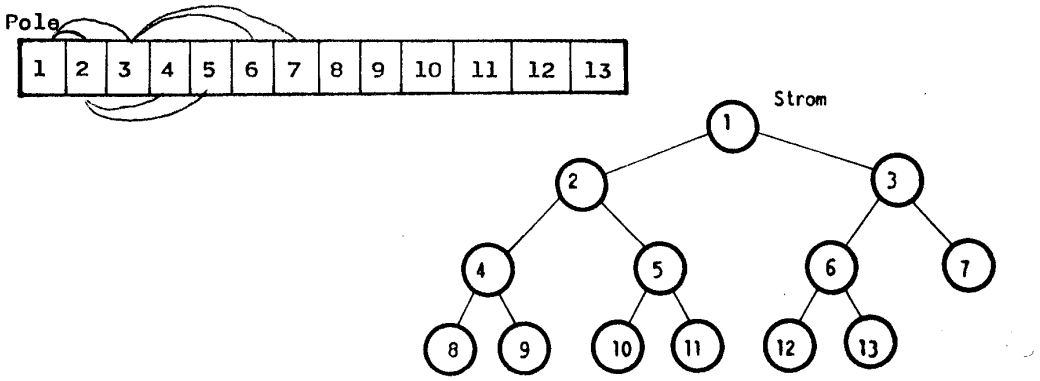
n	256	512
NUP	338	1562
OSP	558	2224

7.2.3. Řazení pomocí stromové struktury (Heap-sort)

Tato metoda je nejvyspělejší metodou postavenou na principu výběru. Zásadním rysem této metody, je způsob vyhledání maxima (resp. minima). Za tím účelem definujme strukturu na základě binárního stromu pro jejíž každý uzel bude platit, že všechny uzly obou podstromů jsou menší (resp. větší) než daný uzel. Takovou stromovou strukturu budeme nazývat hromada (angl. heap, slovensky halda). To znamená, že na vrcholu hromady (v kořeni stromu) je vždy největší (resp. nejmenší) prvek z množiny prvků tvořících hromadu. Jak se ustaví znovu hromada, nahradí-li se vrchol hromady libovolným jiným prvkem? Algoritmu znovu - ustavení hromady se říká "prosetí" (angl. sift) a název charakterizuje proces, v němž se prvek z vrcholu "propadá" až na místo, které mu náleží podle pravidla hromady, zatím co maximum (resp. minimum) "vyplave" na vrchol hromady. Princip řazení hromadou lze vyjádřit takto :

```
ustav hromadu z prvků A[1] až A[N];  {na vrcholu je maximum}
for I:=N downto 2 do
  begin A[I]:=: vrchol hromady;  {t.zn., že pole od I do N je seřazeno}
        Znovu ustav hromadu z prvků A[1] až A[I-1];
  end;
```

Zvláštností této metody je způsob implementace stromu (hromady) v poli (resp. interpretace pole jako stromu), kde synovské uzly jsou svázány s otcovskými implicitními ukazateli (t.zn., že z indexu otce lze spočítat index levého i pravého syna). Je-li i index otcovského uzlu, pak index levého synovského uzlu je $2i$ a pravého synovského uzlu je $2i+1$. Kořen stromu je vždy na indexu 1. Vztah mezi polem s indexy 1 až 13 a stromem je znázorněn na obr. 7.3. Pro strom, kterým se interpretuje pole podle uvedených pravidel musí platit :



Obr. 7.3. Pole interpretované jako binární strom

a) Odeberou-li se od stromu všechny uzly na nejnižší úrovni, je strom absolutně vyvážený a symetrický.
b) Listy na nejnižší úrovni musí být na strom napojeny souvisle zleva doprava. (Chybí-li na nejnižší úrovni ℓ, která má maximálně 2^ℓ uzlů, K uzlů ($K \leq 2^\ell$), pak je to K uzlů zprava. Např. na obr. 7.3. chybí 2 uzly zprava, které by mohly být připojeny k uzlu 7)

Z toho vyplývá, že vrcholem hromady bude v uvedeném algoritmu vždy prvek A[1], a příkaz výměny A[I]:=:vrchol hromady můžeme nahradit příkazem A[I]:=:A[1].

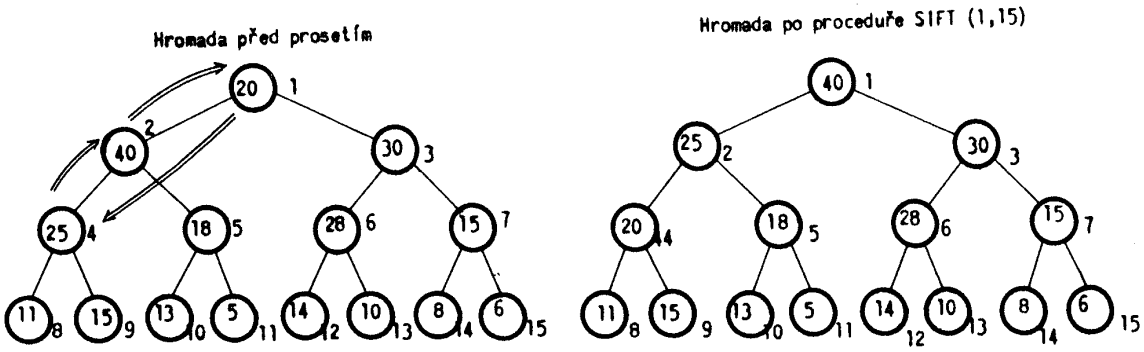
Mechanismus prosetí (znovuustavení hromady) pracuje takto: ze dvou synovských uzlů se pro porovnání s otcem vybere ten, který je lepším kandidátem na vrchol - (tzn. ten ze dvou synovských uzlů, který je větší). Je-li vybraný synovský uzel "lepší" než otcovský (větší), provede se jejich výměna a algoritmus se cyklicky opakuje pro podstrom, jehož kořenem se stal "propadlý" prvek. Cyklus končí nedojde-li k výměně (prvek propadl na správné místo), nebo dojde-li se k listu stromu. Listem stromu implementovaného polem je prvek s indexem K, pro nějž platí $2K > n$. Uzel s indexem K nemá pravého syna, platí-li $(2K+1) > n$.

Procedura, která "proseje" podstrom implementovaný polem prvků mezi indexy L a R (L < R) bude mít tvar :

```
procedure SIFT (L,R:TYPINDEX);  {L je kořen podstromu}
var I,J:TYPINDEX;
    X:integer;
    JESTE:Boolean;
```

```
begin I:=L;
      J:=2*I;  { Index levého syna }
      X:=A[I];
      JESTE:=J ≤ R;
      while JESTE do
        begin if J < R
                then { má levého i pravého syna }
                    if A[J] < A[J+1]
                         then J:=J+1;  { pravý syn je větší }
              if X ≥ A[J]
                  then JESTE:=false  { Prvek X je "proset" na své místo; konec
                                       cyklu }
                  else begin { Prvek X "propadá" níže; Prvek A[J]"vyplave" výše }
                             A[I]:=A[J];
                             I:=J;
                             J:=2*I   { Posun mechanismu na nižší hladinu }
                             JESTE:=J ≤ R  { Test, zda A[I] je list }
                       end
        end; { cyklu }
      A[I]:=X
end;  { procedury SIFT }
```

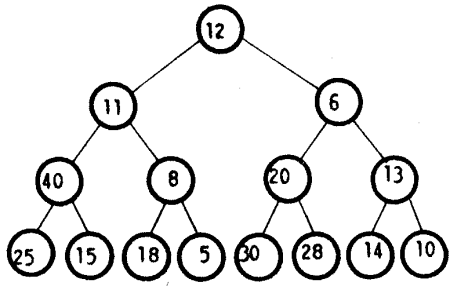
Na obr. 7.4. je znázorněna hromada, jejíž vrchol před prosetím nesplňuje pravidla hromady a hromada po provedení procedury sift. V kroužku jsou hodnoty prvků, vedle kroužku indexy prvků.



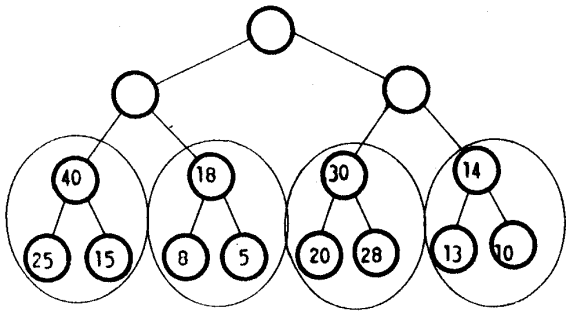
Obr. 7.4. Mechanismus prosetí procedury SIFT

Poslední nevyřešenou otázkou je, jak vytvořit z náhodně uspořádaného pole hromadu. Lze využít toho, že procedura SIFT umí vytvořit hromadu ze stromu sestávajícího ze dvou uzlů (otec a levý syn) a tří uzlů (otec a oba synové). Postup vytvoření hromady pomocí procedury SIFT znázorňuje obr. 7.5.

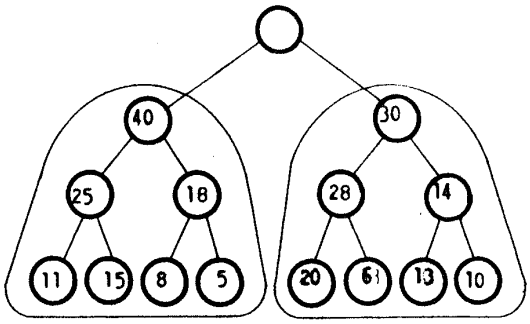
Procedura se vyvolá napřed pro "nejnižšího" a "nejpravějšího" otce. Je-li N index nejpravějšího prvku, pak je to současně syn "nejnižšího" a "nejpravějšího" otce. Tento otec má index I = N div 2. Procedura SIFT se bude tedy dále postupně vyvolávat pro všechny podstromy s indexy menšími než I. Mechanismus vytvoření hromady a seřazení prvku metodou Heapsort popisuje algoritmus :



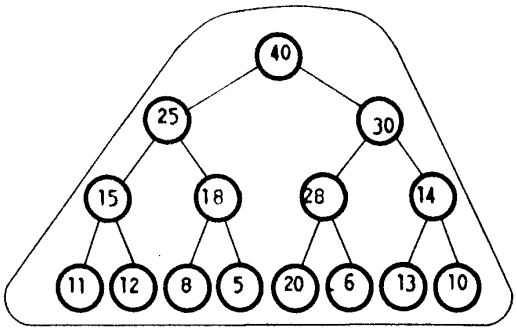
Neuspořádané pole



Procedura sift vytvořila 4 hromady (zprava doleva) na předposlední (3.) úrovni



Procedura sift vytvořila 2 hromady (zprava doleva) na 2.úrovni



V posledním kroku vytvoří procedura ze dvou hromad jedinou hromadu

Obr. 7.5. Postup vytváření hromady

```
L:=N div 2; R:=N;
for L:=L downto 1 do SIFT (L,R);  {Vytvoření hromady}
for R:=N downto 2 do              {Cyklus seřazení metodou Heapsort}
      begin
            A[1]:=: A[R];         {Výměna vrcholu hromady a posledního prvku}
            SIFT (1,R-1)          {Znovuustavení hromady}
      end;
```

Z analýzy metody vyplývají tyto závěry :

a) Metoda **není stabilní** a z dále uvedených výsledků vyplývá, že se **nechová přirozeně**. Metoda pracuje **in situ**. Výsledky jejího časového hodnocení jsou zejména pro větší počet prvků vynikající.

b) Dále uvedené výsledky jsou získány pro poněkud modifikovaný zápis algoritmu. Modifikace spočívá v náhradě cyklů **for** cykly **while**, použitím procedury SIFT bez parametrů a v optimalizaci cyklu v proceduře SIFT s využitím exitového skoku ven z cyklu při podmínce konce "propadání".

c) Pro maximální časovou složitost byl odvozen vztah :

$$T_{MAX}(n) \doteq 31{,}5\ n \lfloor \log_2(n-1) \rfloor + 62.25\ n + 48.5 * 2^{\lfloor \log_2(n-1) \rfloor}$$

d) Pro paměťovou složitost platí vztah :

$$S(n) = 4n + 363$$

e) Experimentálně byly naměřeny tyto hodnoty :

n	256	1024
SP	42	210
NUP	38	186
OSP	40	196

7.3. Řazení na principu vkládání

Princip řazení pomocí vkládání se podobá metodě, jakou hráč karet obvykle seřazuje karty v ruce, když je po rozdání postupně bere ze stolu.

Nechť je pole seřazovaných položek rozděleno na část seřazenou (od indexu 1 do i-1) a na část neseřazených (od i do N). Nalezne-li se index K ($1 \leq K < i$) pro nějž platí $A[K-1] \leq A[i] < A[K]$, pak se část pole od indexu K do (i-1) posune o jednu pozici doprava a na uvolněnou pozici K se vloží zařazovaná položka s indexem i. Inicializace spočívá v rozdělení pole na 2 části: Prvek A[1] tvoří seřazenou část a zbytek pole neseřazenou část. Cyklus zařazování končí zařazením N-tého prvku. Pak algoritmus má obecný tvar :

```
for i:= 2 to N do
  begin  najdi index K, na který se má zařadit prvek A[i];
         posuň část pole od K do (i-1) o jednu pozici doprava;
         vlož na A[K] hodnotu zařazovaného prvku
  end
```

Nejprimitivnější metodu, v níž se odděleně sekvenčně vyhledá index K, potom se posune část pole a nakonec se vloží zařazovaný prvek, nebudeme analyzovat.

7.3.1. Metoda bublinového vkládání (Straight insert-sort)

Tato varianta metody řazení vkládáním kombinuje vyhledávání indexu K s posuvem pole, a pro urychlení algoritmu používá zarážku. Postupné porovnávání zařazovaného prvku s prvky seřazené části pole zprava doleva a posun každého většího prvku pole o jednu pozici doprava se podobá principu bublinového řazení. Zarážku tvoří prvek pole A[Ø], o který musíme rozšířit seřazované pole. Před každým cyklem zařazení se vloží do zarážky hodnota zařazovaného prvku. Pak celý algoritmus metody má tvar :

```
procedure INSERTSORT;
var i,j : Ø..N;
    x:integer;
begin
    for i:=2 to N do
      begin x:=A[i];
            A[Ø]:=x;
            j:=i-1;
            while X<A[j] do {Hledej a posouvej}
              begin A[j+1]:= A[j];
                    j:=j-1
              end;
```

```
            A[j+1]:=x      {Vlož na uvolněné místo}
        end
    end {procedury}
```

Z analýzy metody vyplývají tyto závěry :

a) Metoda je stabilní, chová se přirozeně a pracuje in Situ

b) Pro střední časovou složitost byl odvozen vztah :

$T_s(n) = 3.75\ n^2 + 20.25\ n - 21$

c) Pro maximální časovou složitost byl odvozen vztah :

$T_{MAX}(n) = 7,5\ n^2 + 16.5 - 21$

d) Pro paměťovou složitost platí vztah :

$S(n) = 4n + 160$

e) Experimentálně byly naměřeny tyto výsledky :

n	256	512	1024
SP	4	6	14
NUP	156	614	2330
OSP	312	1262	5008

7.3.2. Metoda vkládání s binárním vyhledáváním (Binary insert-sort)

Tato varianta metody řazení vkládáním používá k vyhledávání pozice pro zařazovaný prvek mechanismus binárního vyhledávání. Binární vyhledávání musí v případě rovnosti klíčů nalézt pozici za nejpravějším ze shodných klíčů. Algoritmus má tvar :

```
procedure BINARYINSERT;
var i,j,m,el,r:0..N;
    x:integer;
begin for i:=2 to n do
        begin x:=A[i];
              el:=1; r:=i-1;  {Nastavení levého a pravého ukazatele}
              while el ≤ r do  {cyklus binárního vyhledání}
                begin m:=(el+r) div 2;
                      if x < A[m] then r:=m-1
                                  else el:=m+1
                end; {cyklu while}
              for j:=i-1 downto el do A[j+1]:=A[j]; {posun části pole}
              A[el]:=x   {zařazení}
        end {cyklu for}
end  {procedury}
```

Z analýzy metody vyplývá :

a) Metoda je stabilní, chová se přirozeně a pracuje in Situ.

b) Pro střední časovou složitost platí :

$T_s(n) \leq 2.25\ n^2 + 20.75\ n + 17\ n \lceil \log_2 n \rceil - 17*2^{\lceil \log_2 n \rceil} - 3$

c) Pro maximální časovou složitost platí :

$$T_{MAX}(n) = 4.5\ n^2 + 18.5\ n + 17(n+1)\ \lfloor \log_2 n \rfloor - 14 - 34*2^{\lfloor \log_2 n \rfloor}$$

d) Pro paměťovou složitost byl odvozen vztah

$$S(n) = 4n + 215$$

e) Experimentálně byly naměřeny tyto výsledky

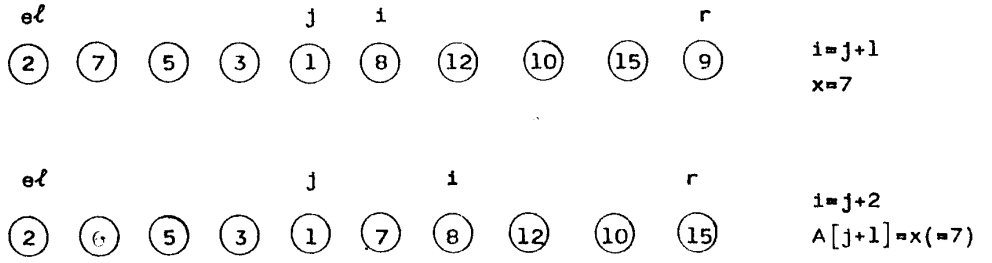
n	256	512	1024
NUP	134	502	1930
OSP	248	956	3736

7.4. Řazení na principu rozdělení (Quick-sort)

Řazení na principu rozdělení je praktickou aplikací zásady "rozděl a panuj", která je jednou z nejvýznamnějších zásad moderní metodiky programování. V obecné rovině pracuje tento princip tak, že množinu seřazovaných prvků M rozdělí na podmnožiny M_1 až M_K, pro které platí $M_1 < M_2 < \ldots < M_K$ (kde $M_i < M_{i+1}$ znamená, že pro každé $x \in M_i$ a $y \in M_{i+1}$ platí $x \leq y$). Mechanismus rozdělení se rekurzivně aplikuje na všechny vzniklé podmnožiny, pokud obsahují alespoň dva prvky, čímž se dosáhne seřazení všech prvků množiny M.

Metoda řazení známá pod názvem Quick-sort je založena na principu rozdělení pole na dvě části M_1 a M_2, pro které bude platit $M_1 < M_2$.

Nechť je dán úsek seřazovaného pole indexem nejlevějšího prvku el a indexem nejpravějšího prvku r. Mechanismus rozdělení musí dosáhnout takového uspořádání prvků úseku pole, že ho lze rozdělit na dva podúseky (A_{el} až A_j) a (A_i až A_r) takové, že pro každé m,n ($el \leq m \leq j$, $i \leq n \leq r$) platí $A_m \leq A_n$. Nechť x je dělicí hodnota, pro kterou platí rovnost předchozího vztahu $x = (A_m = A_n)$ a která se může vyskytovat v obou úsecích pole. Připustíme, aby mechanismus rozdělení skončil dosažením jednoho z těchto stavů indexů



Nechť existuje procedura bez parametrů ROZDĚL, která rozdělí část pole vymezenou indexy el a r na podčásti (el,j) a (i,r) podle uvedených pravidel. (Procedura pracuje s globálními proměnnými el,r,i,j). Pak metodu řazení Quick-Sort lze zapsat touto rekurzivní procedurou :

```
procedure QUICKSORT (el,r:TYPINDEX);
var i,j:TYPINDEX;
begin
      ROZDĚL; {Rozdělení pole na dvě části}
      if el < j then QUICKSORT (el,j); {Rekurzivní volání pro levou podčást}
      if i < r then QUCKSORT (i,r); {Rekurzivní volání pro pravou podčást}
end
```

Zbývá vyřešit dva základní problémy :

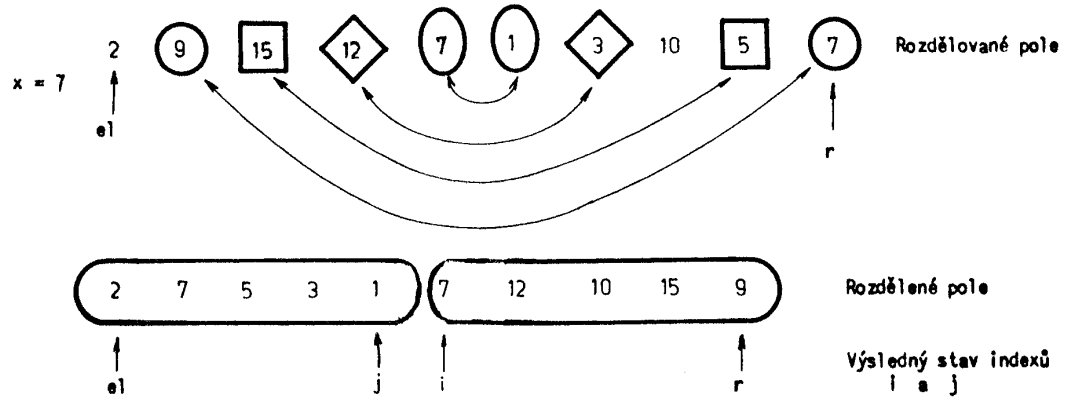
a) jak zvolit dělicí hodnotu x

b) jakým algoritmem co nejefektivněji implementovat proceduru ROZDĚL

Nejvýhodnější dělicí hodnotou x je medián daného souboru prvků (medián daného souboru prvků je hodnota, pro kterou platí, že právě polovina prvků souboru má hodnotu menší a polovina prvků má hodnotu větší). Medián vede k rovnoměrnému rozdělení pole. Jeho určení by však vneslo do algoritmu značné zpoždění, a proto se pro dělicí hodnotu volí určitý prvek pole v naději, že se pomocí něho rozdělí pole na části, které se nebudou svou velikostí výrazně lišit. (Nejhorším případem metody je případ, kdy každé rozdělení rozdělí pole na dva podúseky, z nichž jeden má jen jediný prvek). C.A.R. Hoare v algoritmu publikovaném m.j. v [3] volí pro dělicí hodnotu prostřední prvek pole. Jeho vysoce efektivní a elegantní algoritmus rozdělení má tvar :

```
i:=el; j:=r;
x:=A[(i+j) div 2];  {určení dělicí hodnoty}
repeat
   while A[i] < x do i:=i+1;  {Hledá zleva; nalezne nejblíže vyšší i, pro
                               které platí A[i] ≥ x }
   while A[j] > x do j:=j-1;  {Hledá zprava; nalezne nejblíže nižší j, pro
                               které platí A[j] ≤ x }
   if i ≤ j
        then begin
                  A[i]:=:A[j];
                  i:=i+1;
                  j:=j-1
             end
   until i > j
```

Na obr. 7.6. je znázorněn účinek mechanismu rozdělení.



Obr. 7.6. Účinek mechanismu rozdělení

Algoritmus pracuje tak, že hledá zleva nejblíže další prvek, který je větší nebo roven dělicí hodnotě x a hledá zprava nejblíže další prvek, který je menší nebo roven dělicí hodnotě x. Tyto prvky vzájemně vymění. V této činnosti pokračuje v cyklu tak dlouho, dokud se indexy i a j, postupující proti sobě, "nepřekříží".

Z analýzy Quick-sortu vyplyne :

a) Metoda Quick-sort není stabilní (při vzájemné výměně se může změnit relativní pořadí shodných klíčů). Jak je vidět z experimentů, lze ale odvodit i z principu, metoda nepracuje přirozeně (seřazení náhodně uspořádaného pole je téměř dvojnásobně dlouhé oproti seřazení již seřazeného nebo opačně seřazeného pole). Metoda v důsledku rekurze nepracuje in Situ. Tyto vlastnosti však nebrání tomu, aby metoda byla hodnocena jako jedna z nejúspěšnějších řadicích metod, vhodných pro větší počet prvků. V další části kapitoly bude uvedena modifikace této metody spočívající v nahrazení rekurzivního zápisu algoritmu iterací se zásobníkem.

b) Pro střední časovou složitost byl odvozen vztah :

$T_S(n) < 17\ n \log_2 n - 64$

což je zatím nejúspěšnější (nejmenší) hodnota střední časové složitosti (cca 1,5x rychlejší než Heap-sort)

c) Pro časovou složitost řazení již seřazeného pole platí :

$T_{SP}(n) = 9\ n \log_2 n + 62\ n - 78$

d) Pro časovou složitost řazení opačně seřazeného pole platí :

$T_{OSP}(n) = 9\ n \log_2 n + 72\ n - 141$

e) Pro paměťovou složitost byl odvozen vztah

$S(n) \doteq 17.5\ n + 311$

to znamená prakticky čtyřnásobek všech předchozích metod !

f) Experimentálně byly naměřeny tyto výsledky :

n	256	512	1024
SP	10	24	50
NUP	22	48	102
OSP	12	26	56

7.5. Řazení na principu setřídění (Merge-sort)

Metoda vychází z principu, kterým se setřídí sekvenčně dvě posloupnosti seřazené ve stejném smyslu do jediné posloupnosti seřazené v témž smyslu. Protože se tímto principem pracuje s údaji sekvenčně, je to také základní princip všech vnějších sekvenčních řadicích metod.

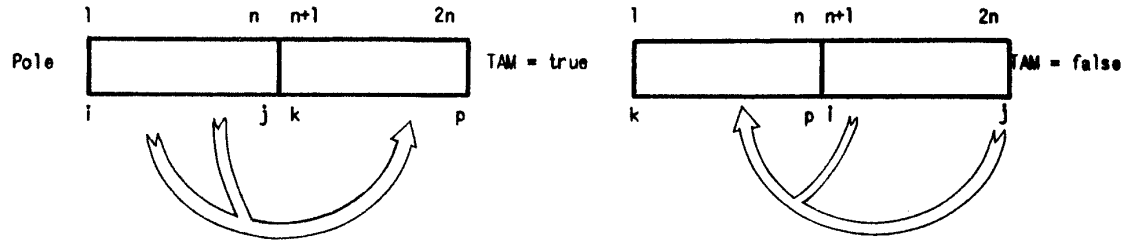
Popišme tento princip na algoritmu setřídění seřazeného pole A:array [1..m] of integer a seřazeného pole B:array [1..n] of integer do výsledného seřazeného pole C:array [1..p] of integer (p=m+n). Algoritmus má tvar :

```
i:=1; j:=1; k:=1;  {Indexy prvků v polích A,B a C}
while (i ≤ m) and (j ≤ n) do  {dokud se nevyčerpá jedno z polí,cykluj}
begin if A[i] < B[j]  {Menší ze dvou vlož do výstupního pole}
        then begin  C[k]:=A[i];
```

```
                    i:=i+1     { nastavení indexu na další prvek v poli A }
              end
         else begin C[k]:=B[j];
                    j:=j+1     { nastavení indexu na další prvek v poli B }
              end;
    K:=K+1
end;
while i ≤ m do begin C[K]:=A[i]; i:=i+1; K:=K+1 end;
  { Zbyla-li část pole A, přepíše se do C }
while j ≤ n do begin C[K]:=B[j]; j:=j+1, K:=K+1 end;
  { Zbyla-li část pole B, přepíše se do C }
```

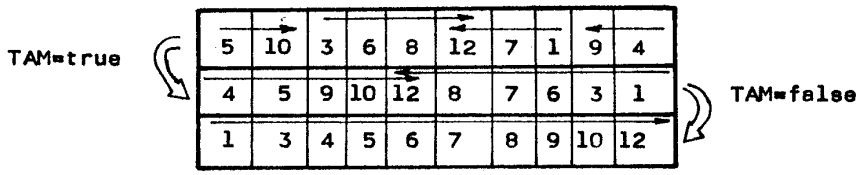
Metoda řazení známá pod názvem Merge-sort využívá principu setřídění a slovně lze její funkci popsat takto : k poli A seřazovaných prvků se připojí stejně dlouhé pole B. Dvojice neklesajících posloupností z levého a z pravého konce pole A se setřídí do jedné neklesající posloupnosti, která se vloží zleva do pole B. V poli A se hledá další dvojice neklesajících posloupností a neklesající posloupnost vzniklá jejich setříděním se tentokráte uloží do pole B zprava. Tímto způsobem se střídavě zleva a zprava dále ukládají další setříděné dvojice tak dlouho, až se naplní pole B (resp. až se protijdoucí indexy posloupností v poli A setkají). Potom si pole A a pole B vymění roli a změní se směr přesunu; pole B je zdrojové a pole A cílové; a celý proces přesunu se opakuje, tak dlouho, až se po jednom přesunu vytvoří v cílovém poli jediná neklesající posloupnost. Je-li výsledná posloupnost v poli B, zkopíruje se zpět do pole A, v němž očekáváme výsledné seřazené pole.

Algoritmus se realizuje ve dvojnásobném poli, v němž první polovina má roli pole A a druhá polovina pole B. Indexy i a j jsou začátek a konec zdrojového pole a indexy k a p začátek a konec cílového pole. Booleovská vyhýbka TAM bude určovat směr přesunu z pole A do B (true) či naopak (false). Situaci v poli znázorňuje obr. 7.7.



Obr. 7.7. Situace při setřiďování ve směrech TAM a not TAM

Postup řazení na principu setřídění je znázorněn na obr. 7.8.



Obr. 7.8. Řazení pole metodou Merge-Sort

Algoritmus bude mít v prvním přiblížení tvar :

```
TAM:=true;  {směr přesunu zleva doprava}
repeat
    {1}  Nastav hodnoty indexů i,j,k,p v závislosti na směru přesunu;
         m:=Ø;  {m je počitadlo posloupností vložených do cílového pole
                 v následujícím abstraktním příkazu}
    {2}  Setřiď všechny dvojice posloupností ze zdrojového pole a vlož
         je do cílového pole;
         TAM:= not TAM;  {inverze směru přesunu}
until m=1;
if notTAM then  {3}  Zkopíruj pravou polovinu pole do levé poloviny;
```

Abstraktní příkaz {1} bude mít po dekompozici při prvním snižování abstrakce tvar :

```
{1}  if TAM then begin i:=1; j:=n; k:=n+1; p:=2*n end
            else begin k:=1; p:=n; i:=n+1; j:=2*n end;
```

Abstraktní příkaz {2} "Setřiď všechny dvojice..." bude mít po dekompozici při prvním snižování abstrakce tvar :

```
{2}  h:=1;  {krok h má hodnotu 1 resp. -1 podle směru postupu posloupnosti}
     repeat
      {2.1}  Setřiď jednu dvojici posloupností ze zdrojového pole od i a j
             do cílového pole od k
             k:=:p;  {Výměna k a p při změně směru cílové posloupnosti}
             m:=m+1;  {zvýšení počtu posloupností přesunutých do cílového
                       pole}
             h:=-h;  {změna polarity kroku při změně směru}
     until i=j;  {dokud se dvě protijdoucí posloupnosti zdrojového pole
                  nesetkají}
```

Abstraktní příkaz {3} pro kopírování bude mít po dekompozici tvar :

```
{3}  for i:=1 to n do  A[i]:=A[i+n]
```

Při druhé úrovni snižování abstrakce bude mít abstraktní příkaz {2.1} "Seřaď jednu dvojici..." po dekompozici tvar :

```
{2.1}  KONECL:=false; KONECP:=false; KONECS:=false;
       repeat
          if A[i]<A[j]    {vyber menší prvek z čela a vlož ho do cíl. pole}
             then begin A[k]:=A[i];
                        if i=j then KONECS:=true {posl. se sešly uprostřed}
                               else begin i:=i+1; k:=k+1;
                                          if A[i]<A[i-1]
                                             then KONECL:=true {konec zleva}
                                    end
                  end
```

```
      else begin A[k]:=A[j];
                 if i=j then KONECS:=true {posl. se sešly uprostřed}
                        else begin j:=j-1; k:=k+h;
                                   if A[j]<A[j+1]
                                     then KONECP:=true {konec posl.zprava}
                                   end
           end
until KONECL or KONECP or KONECS;

 while KONECL do begin  {Skončila levá posl., přidá se zbytek pravé}
                    A[k]:=A[j];
                    j:=j-1;
                    k:=k+h;
                    KONECL:=A[j]≥ A[j+1]
                end;
 while KONECP do begin  {Skončila pravá posl.,přidá se zbytek levé}
                    A[k]:= A[i];
                    i:=i+1;
                    k:=k+h;
                    KONECP:= A[i]≥ A[i-1]
                end
```

Z analýzy metody Merge-sort vyplývá :

a) Metoda není stabilní (postupuje střídavě zleva a zprava a tudíž nerespektuje vzájemné uspořádání shodných klíčů), nechová se přirozeně (algoritmus je stranově symetrický; je-li pole seřazené v kterémkoli směru, "nevyhovuje" to tomu z obou směrů postupu, který je opačný) a nepracuje in situ, protože potřebuje dvojnásobně dlouhé pole, než je seřazované pole.

b) Pro maximální časovou složitost byl odvozen vztah :

$$T_{MAX}(n) = (28\ n + 22)\ \log_2 n + 38\ n + 7$$

c) Pro paměťovou složitost platí vztah

$$S(n) = 8\ n + 704$$

d) Experimentálně byly získány tyto údaje :

n	256	512
SP	8	14
NUP	6	12
OSP	32	72

Tyto výsledky představují jeden z velice rychlých algoritmů.

7.6. Řazení se snižujícím se přírůstkem (Shell-sort)

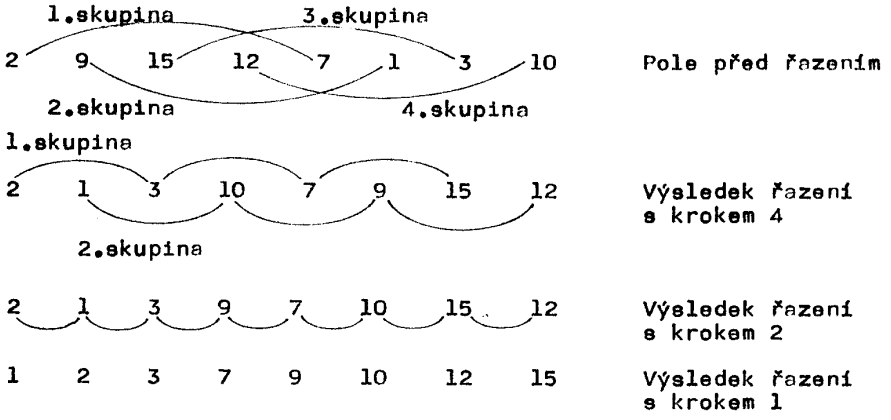
Algoritmus řazení se snižujícím se přírůstkem, označovaný podle jména autora názvem Shell-sort, vzbudil při svém uvedení v 60.letech velkou pozornost. Ještě v publikaci o 10 let později [7] se uvádí : cit."Bohužel, není snadné pochopit činnost této metody. Když se poprvé objevila v tisku, jistý vedoucí 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 z čtenářů domnívá, že by porozuměl Shellově metodě, může to zkusit." Je nutné podotknout, že dále v publikaci uvedený algoritmus vyjádřený vývojovým diagramem je opravdu téměř nepochopitelný do okamžiku, než se přepíše podle zásad strukturovaného programování, kdy je srozumitelný bez komentářů i pro průměrného studenta.

V čem spočívá princip Shellovy metody? Metoda využívá některé z jednoduchých řadicích metod se střední složitostí $O(n^2)$. V těchto metodách se prvky porovnávají a přemisťují vždy o 1 pozici. Lze přitom dokázat, že se každý prvek ve středním případě přesune celkem o n/3 pozic. Mechanismus řazení by se dal urychlit, kdyby se jednoduchá metoda mohla pohybovat v poli většími skoky. Princip, jak toho dosahuje Shellova metoda popíšeme takto :

> Rozdělíme prvky pole na 4 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o krok $h_3=4$ a každou skupinu zvlášť seřadíme vhodnou jednoduchou metodou. V další etapě rozdělíme všechny prvky pole na 2 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o krok $h_2=2$. V poslední etapě seřadíme celou posloupnost všech prvků (s krokem $h_1=1$).

Poslední etapa by sama o sobě seřadila pole prvků a kdyby se jako metoda řazení použila metoda, která se nechová přirozeně, byly by "předzpracující" etapy něčím navíc! Skutečnost však ukazuje, že "předřazovací" etapy urychlí přesun prvků na svá místa a způsobí, že poslední etapa proběhne mnohem rychleji. Postup této metody je pro 8 prvků zobrazen na obr. 7.9.



Obr. 7.9. Princip Shellovy metody

V uvedeném příkladě se v jednotlivých etapách seřazovaly soubory s krokem 4,2 a 1. Shellova metoda nebyla dosud důkladně matematicky analyzovány (a protože byla výrazně překonána metodami s časovou složitostí $O(n \log_2 n)$, praktický zájem o tuto metodu poklesl), a proto není matematicky rozhodnuto, posloupnost jakých snižujících se kroků $h_s, h_{s-1}, \ldots, h_1$ je nejvýhodnější. Mezi nejznámějšími návrhy byly tyto posloupnosti :

A : $h_1=1$, $h_{i+1} = 2h_i + 1$

B : $h_1=1$, $h_2=3$, $h_{i+1}=2h_i-1$ (pro $i>2$)

C : $h_1=1$, $h_{i+1} = 3h_i+1$ (kde nejvyšším použitým h_i je nejmenší h_i, pro něž platí $h_{j+2} \geq n$)

Pro posloupnost B byla dokázána maximální časová složitost $O(n^{1.5})$, což znamená, že Shellova metoda zaujímá pozici mezi pomalými ($O(n^2)$) a rychlými ($O(n \log_2 n)$) metodami řazení.

V následujícím algoritmu je v Shellově metodě použito řazení metodou bublinového vkládání (Straight insertion). Z toho důvodu je nutno rozšířit seřazované pole zleva nejen o prvek A[Ø], ale o h_t prvků, kde h_t je největší použitý krok, pro který platí $h_t < \lfloor \frac{h}{3} \rfloor$ a dále je nutno vytvořit pomocné pole kroku h, do kterého se vloží hodnota kroků h_1 až h_t.

Algoritmus bude tedy pracovat s polem :

```
var A:array [-ht..n] of integer;
```

a pro variantu s posloupností B má tvar :

```
procedure SHELLSORTB;
var i,j,K,s,m,x:integer;
    h:array [1..t] of integer;
begin
   {Generování hodnot pole h}
    j:=n div 3;  h[1]:=1;
    m:=2; i:=3;
    while j > i do begin h[m]:=i; i:=2*i-1; m:=m+1 end;
     {Vlastní řazení}
    m:=m-1;
    while m ≥ 1 do
       begin K:=h[m];   {Ustavení kroku}
             s:=-K;     {index zarážky ve vlevo rozšířeném poli A}
             for i:=K+1 to n do
                begin x:=A[i];
                      j:=i-K;
                      if s=Ø then s:= -K;
                      s:=s+1;
                      A[s]:=x;  {Ustavení zarážky}
                      while x < A[j] do  {cyklus vkládání probubláváním}
                         begin A[j+k]:=A[j];
                               j:=j-K  {snížení indexu o krok}
                         end;
                      A[j+k]:=x
                end; {cyklu for}
             m:=m-1
       end  {cyklu while m ≥ 1}
end  {Procedury}
```

Podobně vypadají procedury SHELLSORTA a SHELLSORTC pro posloupnosti A a C postupně se snižujících přírůstků h.

Z analýzy metody Shell-sort vyplývá :

a) Metoda není stabilní (což je důsledkem kroku většího než 1) a pracuje in situ. Tvrzení o přirozenosti není jednoznačné, jak vyplývá z tabulky experimentálních výsledků.

b) Přesný vztah pro časovou složitost nebyl odvozen. V [2] se uvádí pro SHELLSORTA horní hranice složitosti $T_{MAX}(n)$ řádově $O(n^{1.5})$.

c) Experimentálně byly získány tyto výsledky :

	n	SHELLSORTA	SHELLSORTB	SHELLSORTC
NUP	512	96	96	92
	1024	220	232	222
OSP	512	94	92	66
	1024	212	228	156

7.7. Řazení seznamů

V odst. 7.1.2. byl uveden princip řazení, při němž není nutné přemisťovat jednotlivé položky seřazovaného pole. Používá se při něm pomocné pole ukazatelů na jednotlivé položky. Tento princip je zvláště výhodný tam, kde jsou položky příliš velké a jejich přesunem by se výrazně prodloužil čas potřebný pro seřazení. Výsledkem takového řazení je zřetězený seznam, který lze uspořádat sekvenčně v témže poli.

7.7.1. Řazení tříděním podle základu (Radix-sort)

Metoda řazení tříděním podle základu (také řazení tříděním, angl. Radix-sort) je počítačovou aplikací klasické metody řazení děrných štítků postupným tříděním na třídicích strojích (viz poznámka pod čarou v odst. 7.1.). Ukažme si tento princip na řazení trojciferných čísel desítkové soustavy (základ=10). V průchodu polem roztřídíme všechna čísla do deseti tříd podle hodnoty nejnižší číslice. Každou třídu tvoří fronta čísel (uspořádaná podle příchodu). Pro další průchod se všechny fronty sloučí zřetězením od fronty 0 do fronty 9 a třídění pokračuje ve sloučeném seznamu podle číslice na nejbližším vyšším řádu. Počet třídění je dán maximálním počtem uvažovaných číslic klíče. Tři průchody řazení třímístných čísel znázorňuje obr. 7.10. Jednotlivé fronty jsou označeny šipkami s vyznačenou hodnotou fronty a jejich sloučení je vyjádřeno jejich uvedením za sebou zleva doprava.

008	381	966	377	504	625	199	552	230	416	833	Pole před řazením
230	381	552	833	504	625	966	416	377	008	199	Třídění podle řádu 0
0	1	2	3	4	5	6		7	8	9	
504	008	416	625	230	833	552	966	377	381	199	Třídění podle řádu 1
0		1	2	3		5	6	7	8	9	
008	199	230	377	381	416	504	552	625	833	966	Třídění podle řádu 2
0	1	2	3		4	5		6	8	9	

Obr. 7.10. Řazení trojciferných čísel podle základu 10

Protože fronty budou implementovány zřetězenými seznamy, budeme předpokládat, že kromě seřazovaného pole A bude k dispozici pole ukazatelů stejné velikosti :

```
var P:array [1..N] of integer
```

kde P[i] je ukazatel (index) na prvek pole A, který je následníkem prvku A[i] a který má větší hodnotu číslice klíče. Prvek j s největším klíčem má ukazatel P[j]=maxint.

Dále uvedený algoritmus řazení bude pracovat se základem daným konstantou ZAKLAD, v níž číslice budou mít hodnotu Ø..MAXCIF, kde MAXCIF je konstanta MAXCIF= ZAKLAD-1. Protože chceme, aby algoritmus pracoval i se záporným hodnotami klíčů musíme počet tříd rozšířit o třídy pro číslice záporných čísel. Nezáporné číslice se budou třídit do front (tříd) Ø až MAXCIF podle hodnoty porovnávané číslice, záporná čísla se budou třídit do front -MAXCIF až -1 podle hodnoty -(číslice + 1), kde "číslice" je porovnávaná číslice záporného čísla.

Definujme pole ukazatelů na začátky a konce všech front, které jsou jednosměrně zřetězené od konce k začátku :

```
KONFRONTY:array [-ZAKLAD..MAXCIF] of integer;
ZACFRONTY:array [-ZAKLAD..ZAKLAD] of integer;
```

Je-li fronta i prázdná pak bude ukazatel ZACFRONTY[i]=maxint a ukazatel KONFRONTY[i] je nedefinovaný. Ukazatel "navíc" - s indexem ZAKLAD - ZACFRONTY [ZAKLAD]=maxint - plní funkci zarážky.

Celý algoritmus má při prvním přiblížení tento tvar :

```
begin
 {1} Inicializace pomocných proměnných
     Stanovení hodnoty POCCIF - maximálního počtu číslic klíče;

     for j:=1 to POCCIF do {Řazení postupným tříděním pro POCCIF číslic}
       begin
         {2} Inicializace pole ZACFRONTY;

         {3} Třídění do front podle j-té číslice;

         {4} Nalezení nejnižší neprázdné fronty (v poli ZACFRONTY)
             a uložení jejího ukazatele do UKMIN;

         {5} Spojení front do jediného seznamu počínaje prvkem A[UKMIN]
       end;
 {6} Sekvenční seřazení prvků seznamu do pole.
end
```

Po dekompozici abstraktních příkazů dostaneme :

```
{1} {Inicializace pom.proměnných. Stanovení max.počtu číslic - POCCIF.}
    RAD:=1;   {RAD je hodnota 10^r, kde r je řád zkoumané číslice}
    POM:=n;   {Pomocný ukazatel prvků}
    ZACFRONTY [ZAKLAD] :=maxint; {Ustavení zarážky}
    MAX:=abs(A[1]);   {MAX je pomocná proměnná pro hledání
                                          "nejdelšího" klíče}
    for i:=2 to n do if MAX < abs(A[i])
                     then MAX:=abs(A[i]);
    POCCIF:=Ø;   {POCCIF je maximální počet číslic}
    while MAX≠Ø do
          begin MAX:=MAX div ZAKLAD;
                POCCIF:=SUCC(POCCIF)
          end;
```

{2} Inicializace pole ZACFRONTY

```
for i:=-ZAKLAD to MAXCIF do
        ZACFRONTY[i]:=maxint;
```

{3} Třídění do front podle j-té číslice

```
KONEC:=false;   {řídicí proměnná cyklu repeat}
repeat
   {Určení hodnoty číslice na řádu RAD a její uložení do proměnné CIF}
    if A[POM] ≥ Ø
         then CIF:= (A[POM] mod (ZAKLAD*RAD)) div RAD
         else CIF:=-(abe(A[POM])mod (ZAKLAD*RAD)) div RAD-1;
         {Zařazení prvku A[POM] do fronty CIF}
         if ZACFRONTY[CIF]=maxint
              then begin   {Fronta je prázdná}
                           ZACFRONTY[CIF]:=POM;
                           KONFRONTY[CIF]:=POM
                   end
              else begin  {Fronta je neprázdná}
                           P[KONFRONTY[CIF]]:=POM;  {Ukazatel posledního se
                                                     nastaví na vložený prvek}
                           KONFRONTY[CIF]:=POM      {Konec fronty ukazuje
                                                     na vložený prvek}
                   end;
  {Příprava ukazatele POM a ustavení konce cyklu}
    if j=1
        then if POM=1 then KONEC:=true
                      else POM:=POM-1   {Při prvním průchodu se prochází
                                         polem sekvenčně}
        else begin  {Pro ostatní průchody se prochází zřetězeným seznamem}
                    POM:=P[POM];
                    if POM=maxint then KONEC:=true
             end
until KONEC
```

{4} Nalezení ukazatele UKMIN na nejnižší neprázdnou frontu
(a také příprava řádu RAD na další průchod)

```
RAD:=RAD*ZAKLAD;   {Příprava řádu - zvýšení}

i:=-ZAKLAD; UKMIN:=i;
while ZACFRONTY[i]=maxint do  {Hledání nejnižší neprázdné fronty}
         i:=i+1;
UKMIN:=i;
```

{5} Spojení front do jediného seznamu

```
repeat
POM:=KONFRONTY[i];
i:=i+1;
while (i ≠ ZAKLAD) and (ZACFRONTY[i]=maxint do i:=i+1;
                        {Přeskočení prázdných front}
P[POM] :=ZACFRONTY[i]   {Zřetězení konce fronty i na začátek fronty
                         i+1}
until i=ZAKLAD;
POM:=ZACFRONTY[UKMIN];
```

{6} Mac Làren-ův algoritmus pro sekvenční uspořádání zřetězeného seznamu v poli, bez pomocného pole

```
i:=1;
while i < n do
  begin
       while POM < i do POM:=P[POM];
       A[POM]:=:A[i]; P[POM]:=:P[i];
       P[i]:=:POM;
       i:=i+1
  end
```

Z analýzy algoritmu vyplývá :

a) Metoda je stabilní, stav uspořádání pole nemá podstatný vliv na časovou složitost algoritmu a metoda se tedy nechová přirozeně a nepracuje in situ.

b) Pro časovou složitost byl odvozen vztah založený na předpokladu, že pole je seřazeno před řazením opačně, že klíče jsou záporné a že žádná fronta nezůstane prázdná (což je reálný přepoklad pro dostatečně velká n). Pak

$$T_{MAX}(n) \approx (42 * POCCIF + 15) * n + (16 + 34 * ZAKLAD) * POCCIF + 15$$

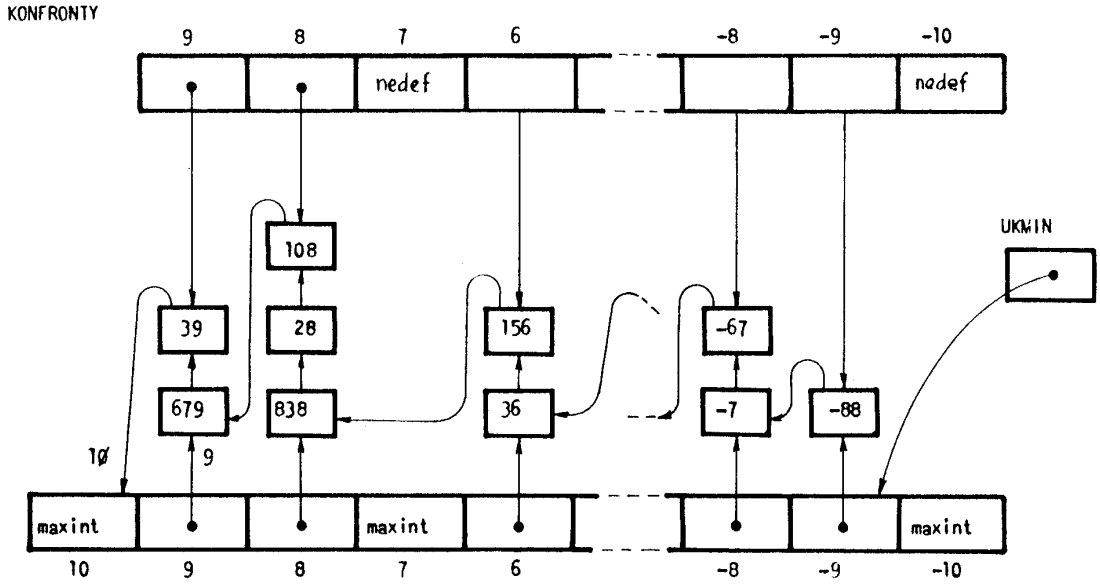
Ze vztahu je vidět, že jde zatím o nejrychlejší algoritmus s lineární časovou složitostí O(n)!

c) Experimentálně byly dosaženy tyto výsledky (pro klíč typu integer na 4 bytech) pro náhodně uspořádané pole (NUP)

n	256	512	1024
Radix-sort	24	60	102

Výhoda oproti srovnatelnému Quick-sortu se zvýší při vyšších n a především při velkých položkách řazeného pole.

Datová struktura a její stav po třídění posloupnosti čísel : 36, 679, 838,28, -88, 39, -7, 108, -67, 156 je zobrazena na obr. 7.11. (pro třídění podle nejnižší číslice).



Obr. 7.11. Datová struktura po sjednocení front vzniklých tříděním podle nejnižší číslice

7.7.2. Řazení setřiďováním seznamů (List-Merge-Sort)

Metoda řazení setřiďováním seznamů využívá principu přirozeného seřazování setřiďováním (Merge-Sort), ale s jednotlivými prvky pole pracuje jako s prvky jednosměrně vázaného seznamu. Slovně lze algoritmus popsat takto :

a) V jednom průchodu polem zřetězíme všechny po sobě jdoucí prvky vytvářející neklesající posloupnosti. Ukazatel posledního prvku nastavíme na koncovou hodnotu (např. Ø) a indexy všech prvních prvků posloupností vložíme do "zdrojové" fronty QZDR.

b) Z každé dvojice po sobě jdoucích seznamů (neklesajících posloupností), jejichž začátky získáme vyjmutím dvou prvků ze zdrojové fronty QZDR, vytvoříme jeden seznam (koncový ukazatel seznamu opět nastavíme na koncovou hodnotu), index začátku tohoto seznamu vložíme do cílové fronty QCIL a zvýšíme počitadlo seznamů, jejichž začátky jsme vložili do QCIL.

Je-li v QZDR jen jeden prvek (index posledního seznamu), nesetřiďujeme jej, index jeho začátku vložíme do QCIL se současným zvýšením počitadla seznamů v QCIL

Až vyprázdníme frontu QZDR, zaměníme mezi sebou funkce front QZDR a QCIL a opakujeme setřiďování ze (změněného) QZDR do (změněného) QCIL tak dlouho, dokud nedosáhneme toho, že v QCIL je jediný prvek - index začátku jediné neklesající posloupnosti.

V prvním přiblížení lze metodu LISTMERGESORT popsat takto :

```
INITQUE (QZDR); INITQUE (QCIL);
{1} Při průchodu polem zřetězíme prvky neklesajících posloupností;
    zakončí každou posloupnost koncovým ukazatelem;
    vlož index začátku každé posloupnosti do QZDR
```

```
repeat
    n:=Ø;   {Nulování počitadla posloupností v QCIL}
      while not EMPTYQUE(QZDR) do
        begin REMOVE(QZDR,ZAC1);  {Čtení do ZAC1 a odstranění z fronty}
            if EMPTYQUE(QZDR)
                then begin    {ZAC1 byl poslední ve frontě QZDR}
                          QUEUP(QCIL,ZAC1);  {Vložení do QCIL}
                          n:=n+1    {Aktualizace počitadla}
                     end
                else  {ZAC1 nebyl poslední; budou se setřiďovat dva}
                     begin REMOVE(QZDR,ZAC2);
                          if A[ZAC1] < A[ZAC2] then ZAC:=ZAC1
                                               else ZAC:=ZAC2;
                          QUEUP(QCIL,ZAC); {Vložení nového začátku
                                            do QCIL}
                          n:=n+1;
                  {2}  Setřiď seznamy ZAC1 a ZAC2; nastav koncový
                       ukazatel posloupnosti
                     end
        end; {cyklu while; konec přesunů z QZDR do QCIL}
    {3}  Zaměň mezi sebou význam QCIL a QZDR
  until n=1;  {Ve frontě - po záměně zdrojové - je jen jedna posloupnost}
{4}  Sekvenční umístění zřetězeného seřazeného seznamu v poli.
```

Protože popsaný algoritmus neobsahuje žádné složité obraty, ponecháme zbývající dekompozici abstraktních příkazů {1}, {2}, {3}, {4} i způsob implementace operací ATD fronta na čtenáři.

Z analýzy algoritmu vyplývá :

a) Algoritmus není stabilní a nepracuje in situ. Podle vztahu doby seřazení již seřazeného a náhodně uspořádaného pole se metoda chová přirozeně. Doba pro seřazení OSP je však menší než pro NUP. Délka řazení není závislá na délce položky.

b) Experimentálně byly naměřeny tyto výsledky :

n	256	512
NUP	32	74
OSP	22	48

Experiment byl proveden na programu, v němž byla fronta a její operace implementovány dynamicky. Operace nad frontou nebyly implementovány jako samostatné procedury.

7.8. Modifikace některých řadicích algoritmů

V tomto odstavci se stručně zmíníme o některých modifikacích řadicích algoritmů. Bude na nich vidět, že časovou složitost lze zlepšit za cenu zhoršení paměťové složitosti. Podrobněji uvedeme pouze významnější algoritmy.

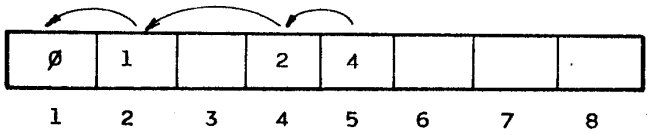
7.8.1. Modifikace přímého výběru (Modif.-Selection-Sort)

Existuje řada méně významných modifikací principu výběru jejichž zobecněním je metoda Heap-sort. Zde se zmíníme o modifikaci přímého výběru. Tato metoda mnohem více využívá informace, kterou lze získat ze srovnání klíčů při hledání maxima. Získává informaci o tzv. levostranných maximech, ukládá ji do pomocného pole a využívá ji ke zkrácení hledání dalšího maxima.

Algoritmus postupuje takto : do pomocné proměnné x se vloží klíč prvního prvku a první prvek pomocného pole se vynuluje. Nalezne-li se při průchodu polem prvek s klíčem větším než x, pak jeho hodnotu vložíme do x a do pomocného pole s týmž indexem vložíme index předchozího maxima. V další etapě budeme pole prohledávat až od předchozího maxima. Mějme např. pole klíčů

25 (43) 19 (63) (92) 87 21 38

pak zakroužkovaná čísla jsou levostranná maxima, ze kterých v pomocném poli M:array [1..n] of Ø..n vznikne tato struktura :



Po výměně nalezeného maximálního prvku s posledním prvkem se testuje, zda předchozí levostranné maximum je větší než vyměněný prvek. Když není, tak se dále bude vyhledávat od pozice posledního maxima, jinak se bude vyhledávat od pozice předchozího levostranného maxima.

```
procedure MODSELECTSORT
var i,j,p,k,x: integer;  {p je index posledního maxima, k je index od kterého
                          se začne prohledávat, x je pomocná proměnná
                          pro výměnu prvku pole A}
    M:array [1..n] of Ø..n;
begin
    k:=1; p:=1; i:=n;  {Inicializace pomocných proměnných}
    M[1]:=Ø;     {Inicializace pomocného pole}
    x:=A[1];
    while i >= 2 do   {Cyklus řazení hledá maxima a ukládá je zprava}
       begin for j:=k+1 to i do   {Hledání levostranných maxim}
                if A[j] > x
                   then begin x:=A[j];  {Nové levostranné maximum}
                              M[j]:=p;  {Ukazatel na předchozí levostr.max}
                              p:=j      {Aktualizace p}
                        end;
```

```
            while (p=i) and (p≠1) do
             {Je-li p=i, pak je maximum již na svém místě. Není-li p=1
              snížíme i o 1 }
            begin i:=i-1; p:=M[p]; x:=A[p]
            end;
          A[p]:=A[i];     {Uvolnění místa pro maximum }
          A[i]:=x;        {Vložení maxima na začátek seřazené části}
          i:=i-1;         {rozšíření seřazené části pole }
          k:=p;           {Nastavení nového začátku prohledávání }
          if M[p] ≠Ø
              then if A[p]< A[M[p]]
                      then p:=M[p]; {Korekce nového začátku prohledávání }
          x:=A[p]
      end  {cyklu řazení }
end  {procedury MODSELECTSORT }
```

a) Metoda není stabilní, chová se přirozeně a nepracuje in situ. Její časová složitost je typu $O(n^2)$ stejně jako její původní varianta, má však menší multiplikativní konstantu a je proto 2-3x rychlejší.

b) Pro srovnání uvedeme experimentální výsledky současně s hodnotami metod SELECTSORT a BUBBLESORT pro délku pole n=512.

n=512	SELECTSORT	BUBBLESORT	MODSELECTSORT
SP	738	4	10
NUP	744	1562	370
OSP	968	2224	374

7.8.2. Modifikace bublinového výběru (Shaker-sort)

O principu této varianty bublinového výběru jsme se již zmínili v odst. 7.2.2.. Z úvahy, že zatímco při průchodu polem zleva doprava se maximum "probublá" až na konec neseřazené části pole a minimum se posune pouze o 1 pozici doleva směrem ke svému konečnému místu, dochází se k závěru, že by se algoritmus stranově symetrizoval. Pak jeden průchod půjde zleva doprava, další zprava doleva a směr průchodů se střídá a v levé i v pravé části pole se na koncích vytváří seřazené úseky pole. Protože algoritmus lze snadno odvodit z algoritmu v odst. 7.2.2., nebudeme jeho rozpis uvádět.

Z analýzy metody vyplývá :

a) Metoda je stabilní, chová se přirozeně a pracuje in situ.

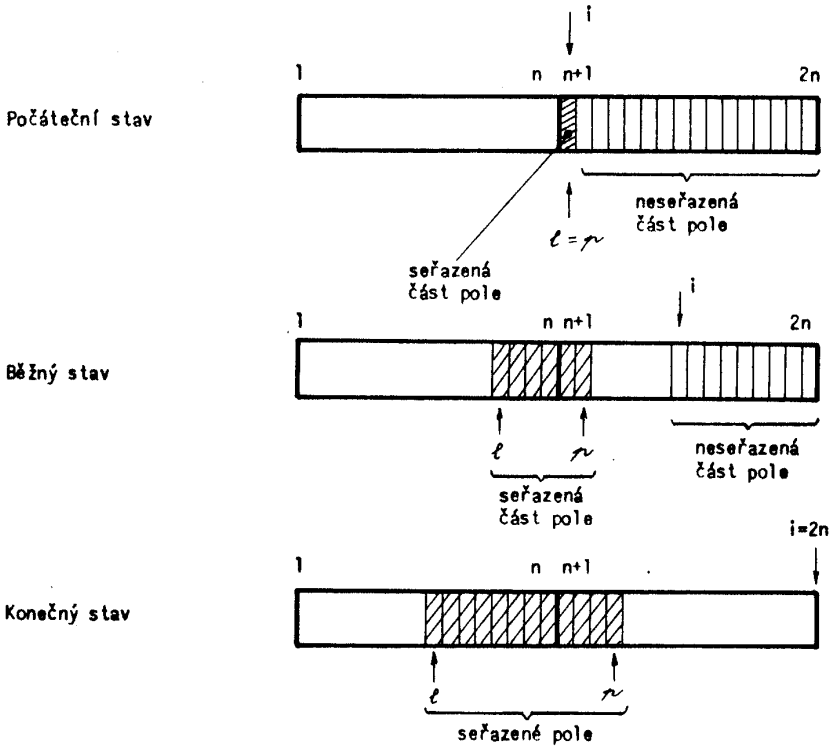
b) Experimentálně byly získány výsledky, které pro n=256 srovnáme s původní metodou

n=256	BUBBLESORT	SHAKERSORT
SP	2	2
NUP	388	340
OSP	588	588

Z výsledků je zřejmé nevýrazné zlepšení pro případ náhodně uspořádaného pole.

7.8.3. Modifikace binárního vkládání (Modif.-Binary-insert-sort)

Jednou z možností, jak zrychlit metodu binárního vkládání, je snížit počet přesunů při posunu části pole, který uvolňuje místo pro vložení prvku, nalezené binárním vyhledáváním. Za cenu dodatečné pomocné paměti lze po určení, zda je nalezené místo blíže levému či pravému konci seřazené části pole, posunovat tu část pole, která je kratší. Za tím účelem potřebujeme dvojnásobné pole a seřazovaná část se umístí do jeho pravé poloviny. Seřazená část pole se bude vytvářet vlevo od původně vloženého pole a bude ohraničována levým a pravým ukazatelem. Situaci v průběhu řazení znázorňuje obr. 7.12.



Obr. 7.12. Průběh řazení modifikovaným binárním vkládáním

Experimentálně získané výsledky porovnáme s původní metodou

	N U P			O S P		
n	256	512	1024	256	512	1024
Binary-insert-sort	134	502	1930	248	956	3736
Modif.-Binary-insert-sort	92	336	1202	24	56	120

Z tabulky je zřejmé výrazné zrychlení pro opačně seřazené pole. U náhodně uspořádaného pole je zrychlení podstatně menší a nesmíme opomenout, že bylo získáno za cenu dvojnásobných paměťových nároků.

7.8.4. Modifikace Quick-sortu

Z hlediska praktického využití jedné z nejrychlejších metod - Quick-sortu - je významnou modifikací odstranění rekurze se současnou redukcí paměťové složitosti.

Mechanismus rozdělení rozdělí daný úsek pole na dva podúseky. Pro jeden z nich se může v iteraci znova vyvolat mechanismus rozdělení, zatímco hraniční indexy druhého se uchovají v zásobníku pro pozdější zpracování. Místo abychom systematicky levý podúsek dělili a pravý uchovávali v zásobníku (jak to dělá algoritmus uvedený v [3]), budeme dělit vždy menší podúsek, zatímco meze většího uchováme v zásobníku. Jakmile má menší z podúseků méně než 2 prvky, vybereme úsek pro další dělení ze zásobníku. V dělení pokračujeme, pokud je zásobník neprázdný. Bude-li se dělit vždy menší podúsek, pak v nejhorším případě (budou-li oba podúseky vždy stejně dlouhé) bude potřebná kapacita zásobníku $k=\log_2 n$. Celý algoritmus modifikovaného nerekurzívního Quick-sortu má po úpravách podle [3] tvar :

```
procedure MODQUICKSORT;
const K = {hodnota určená ze vztahu k=log2n}
var i,j,ℓ,r : Ø..n;  {Indexy a pomocné ukazatele}
    x:integer;   {Pom.proměnná téhož typu jako prvky pole}
    s:Ø..K;      {Ukazatel vrcholu zásobníku}
    STACK:array [1..K] of
                record
                    levy,pravy:1..n
                end;
begin
      s:=1;    {Inicializace zásobníku}
      STACK[S].levy:=1;
      STACK[S].pravy:=n;
      repeat   {zpracuj interval z vrcholu zásobníku}
            ℓ:=STACK[S].levy; r:=STACK[S].pravy; S:=S-1;
            repeat {Rozděl interval A[ℓ]..A[r]}
               i:=ℓ; j:=r; x:=A[(ℓ+r) div 2];
               repeat
                     while A[i]<x do i:=i+1;
                     while x<A[j] do j:=j-1;
                     if i≤j then begin A[i]:=:A[j];
                                       i:=i+1;
                                       j:=j-1
                                 end
               until i>j;
               if (j-ℓ)<(r-i)
                  then begin {levý podúsek je menší}
                         if i<r then
                            begin {vložení pravého podúseku do zásobníku}
                                  S:=S+1;
                                  STACK[S].levy:=i; STACK[S].pravy:=r
                            end;
                         r:=j
                       end
```

```
        else begin
               if j>l then {vložení levého podúseku do zásobníku}
                          begin S:=S+1;
                                STACK[S].levy:= l ;
                                STACK[S].pravy:=j
                          end;
             end l:=i
      until l≥r {Konec rozdělování, bude se brát nový podúsek
                 ze zásobníku}
    until S=Ø  {zásobník je prázdný, řazení je ukončeno}
end {procedury}
```

Základní vlastnosti tohoto algoritmu jsou stejné jako pro rekurzivní verzi. Pro paměťovou složitost však platí :

$$S(n) = 4n+2\log_2 n+8+C \qquad (C \text{ je paměť potřebná pro program})$$

Z tohoto vztahu vyplývá, že nerekurzivní verze s volbou děleného podúseku je paměťově výrazně výhodnější než rekurzivní verze.

V následující tabulce porovnáme experimentálně získané výsledky rekurzivní i nerekurzivní verze Quick-sortu.

	N U P			O S P		
n	256	512	1024	256	512	1024
QUICKSORT	12	26	56	22	48	102
MODQUICKSORT	18	28	60	20	50	106

Výsledky nerekurzivní verze jsou při značné úspoře paměti nevýznamně horší.

7.9. H o d n o c e n í m e t o d v n i t ř n í h o ř a z e n í

V tabulkách tab. 7.2., 7.3. a 7.4. jsou shrnuty výsledky všech experimentů s uvedenými metodami vnitřního řazení (viz [1]). Časové údaje jsou v 10^{-2}s. Algoritmy byly zapsány v jazyce Pascal a odladěny na počítači EC 1033 s kompilátorem STONY BROOK v r.1980. Parametr kompilátoru DEBUG=Ø. Bylo prověřeno, že časy získané experimenty s týmž programem se v důsledku přerušení systémovými programy liší bezvýznamně. Byla-li naměřena hodnota menší než 10 ms, je v tabulce uvedena hodnota Ø.

V tab. 7.5. je pro některé metody uvedena střední míra zpomalení. Řadíme-li soubor o n prvcích jistou metodou jednou pro délku prvku pl a podruhé pro délku prvku p2, pak míra zpomalení je poměr časů T_{p2}/T_{p1} potřebných k seřazení pole prvků. Střední míra se získá z těchto poměrů pro n v rozsahu od 64-1024. Údaje v tabulce indikují míru zpomalení procesu řazení při délce prvku 64 bytů oproti řazení prvků s délkou 4 byty. Velké zpomalení signalizuje vhodnost aplikace metody bez přesunu prvků.

Algoritmy řazení můžeme podle časové složitosti rozdělit do tří skupin :
pomalé algoritmy s časovou složitostí $O(n^2)$
rychlé algoritmy s časovou složitostí $O(n \log_2 n)$
velmi rychlé algoritmy s časovou složitostí $O(n)$

Lineární složitost velmi rychlých algoritmů je podmíněna dodatečnou informací o struktuře klíčů. Na znalosti této struktury je postaven algoritmus (Radix-sort).

Při volbě řadicí metody lze hodnotit tato kriteria :

1. Pomalé algoritmy jsou vhodné pro malý počet řazených prvků (n).
   a) Je-li kritický čas a nezáleží-li na spotřebě paměti, doporučuje se použít modifikace binárního vkládání (MODBINARYINSERT) a modifikace přímého výběru (MODSELECTSORT). Je možné použít také binárního vkládání (BINARYINSERT) nebo bublinového vkládání (INSERTSORT), protože jsou velmi jednoduché.
   b) Je-li kritická spotřeba paměti, postačí binární vkládání (BINARYINSERT) nebo bublinové vkládání (INSERTSORT).

V každém případě se vyhýbáme praktickému použití bublinového výběru (BUBBLESORT) a jeho modifikacím (SHAKERSORT aj.). Jsou zajímavé pouze teoreticky a z hledisek výuky algoritmizace.

2. Pro velký počet řazených prvků volíme některou z rychlých nebo velmi rychlých metod.
   a) Je-li délka prvku pole malá, pak :
   A) Je-li kritickou veličinou čas, lze doporučit rychlé řazení (QUICKSORT) nebo jeho nerekurzívní verzi (MODQUICKSORT). Je-li pravděpodobnost, že pole není náhodně uspořádané vysoká, lze zvolit vhodnější způsob volby "pseudomediánu" (pomocí něhož se úsek pole dělí na dvě části), nebo raději zvolit algoritmus na principu setřídění (MERGESORT).
   B) Je-li kritickou veličinou spotřeba paměti, pak je vhodná nerekurzívní verze rychlého řazení (MODQUICKSORT) nebo řazení hromadou (HEAPSORT).
   b) Je-li délka prvku pole velká, pak se doporučují metoda na principu setřiďování seznamů (LISTMERGESORT) a metoda řazení podle základu (RADIXSORT). Metoda RADIXSORT je nejvhodnější metodou pro délku prvku $k \geq 64$ a dostatečně velký počet prvků ($n \geq 1024$).

č.	n →	64		128		256		512		1024	
	Algoritmy	d1	d2	d1	d2	d1	d2	d1	d2	d1	d2
1	SELECTSORT	14	26	54	58	192	204	738	990	2922	3644
2	BUBBLESORT	0	2	0	0	2	2	4	4	8	-
3	HEAPSORT	8	12	20	28	42	62	94	138	210	304
4	INSERTSORT	0	2	2	6	4	10	6	18	14	36
5	BINARYINSERT	4	8	12	14	22	26	52	58	114	120
6	QUICKSORT	2	4	6	10	10	14	24	30	50	68
7	SHELLSORTB	6	-	14	-	32	-	76	-	182	-
8	SHELLSORTA	4	-	10	-	32	-	70	-	152	-
9	SHELLSORTC	4	-	10	-	18	-	46	-	94	-
10	MODSELECT-SORT	2	2	4	6	6	10	10	18	20	34
11	MODBINARY-INSERT	6	-	16	-	28	-	64	-	138	-

Tab. 7.2. (pokračování na str. 240)

Tab. 7.2. - pokračování

12	SHAKERSORT	0	0	2	2	2	2	2	4	6	-
13	MODQUICK-SORT	2	4	8	8	10	14	24	30	54	62
14	MERGESORT	0	-	6	-	8	-	14	-	28	-
15	RADIXSORT	4	-	16	-	28	-	50	-	148	-
16	LISTMERGE-SORT	2	-	2	-	2	-	6	-	10	-

Tab. 7.2. Rychlost algoritmů řazení pro seřazené pole (SP)

V tab. 7.2.-7.4. je sloupec d1 pro délku prvku 4 byty, d2 pro 64 byty.

č.	n →	64		128		256		512		1024	
	Algoritmy	d1	d2	d1	d2	d1	d2	d1	d2	d1	d2
1	SELECTSORT	16	26	50	62	212	298	744	976	2942	3860
2	BUBBLESORT	26	42	94	166	388	686	1562	2968	6860	-
3	HEAPSORT	8	12	16	26	40	66	88	130	198	288
4	INSERTSORT	12	18	40	68	156	274	614	1080	2330	4234
5	BINARYINSERT	12	20	38	62	134	236	502	910	1930	3722
6	QUICKSORT	4	8	10	16	22	38	48	78	102	166
7	SHELLSORTB	10	-	20	-	44	-	96	-	232	-
8	SHELLSORTA	6	-	18	-	42	-	96	-	220	-
9	SHELLSORTC	8	-	14	-	34	-	92	-	222	-
10	MODSELECT-SORT	8	12	26	32	94	108	374	418	1450	1592
11	MODBINARY-INSERT	8	-	26	-	92	-	336	-	1202	-
12	SHAKERSORT	22	36	78	146	340	606	1454	2448	5408	-
13	MODQUICK-SORT	6	6	14	16	20	34	50	70	106	154
14	MERGESORT	6	-	18	-	32	-	72	-	182	-
15	RADIXSORT	8	-	12	-	24	-	60	-	102	-
16	LISTMERGE-SORT	8	-	14	-	32	-	74	-	150	-

Tab. 7.3. Rychlost algoritmů řazení pro náhodně uspořádané pole (NUP)

č.	n →	64		128		256		512		1024	
	Algoritmus	d1	d2	d1	d2	d1	d2	d1	d2	d1	d2
1	SELECTSORT	20	42	64	100	254	468	968	1796	3856	7008
2	BUBBLESORT	36	72	140	288	558	1158	2224	4634	9866	-
3	HEAPSORT	6	12	18	26	38	56	84	124	186	272

Tab. 7.4. (pokračování na str. 241)

Tab. 7.4. - pokračování

4	INSERTSORT	22	40	80	136	312	538	1262	2140	5008	8556
5	BINARYINSERT	18	32	66	118	248	452	956	1768	3736	7366
6	QUICKSORT	4	6	6	10	12	18	26	38	56	82
7	SHELLSORTB	6	-	16	-	40	-	92	-	228	-
8	SHELLSORTA	6	-	16	-	42	-	94	-	212	-
9	SHELLSORTC	8	-	12	-	30	-	66	-	156	-
10	MODSELECT-SORT	8	10	26	32	96	110	370	408	1456	1592
11	MODBINART-INSERT	6	-	12	-	24	-	56	-	120	-
12	SHAKERSORT	40	72	158	288	588	1152	2496	4612	9880	-
13	HODQUICK-SORT	2	6	6	10	18	18	28	36	60	78
14	MERGESORT	2	-	4	-	6	-	12	-	26	-
15	RADIXSORT	10	-	16	-	36	-	76	-	144	-
16	LISTMERGE-SORT	4	-	10	-	22	-	48	-	100	-

Tab. 7.4. Rychlost algoritmů řazení pro opačně seřazené pole (OSP)

Algoritmus	SP	OSP	NSP
SELECTSORT	1.32	1.84	1.38
BUBBLESORT	1	2.05	1.73
MEAPSORT	1.46	1.57	1.54
INSERTSORT	2.77	1.75	1.71
BINARYINSERT	1.3	1.84	1.76
QUICKSORT	1.54	1.52	1.72
MODSELECTSORT	1.53	1.17	1.22
SHAKERSORT	1.25	1.86	1.74
MODQUICKSORT	1.36	1.65	1.40
RADIXSORT	1	1	1
LISTMERGESORT	1	1	1

Tab. 7.5. Střední míra zpomalení pro prvky s 64 byty oproti prvkům se 4 byty

Poznámka : jestli $T_4 < 10$ ms a $T_{64} < 10$ ms, pak předpokládáme, že míra zpomalení je 1.

Tabulka tab. 7.6. dokládá, jak si odpovídají teoreticky získané hodnoty časové složitosti s experimentálně získanými metodami. Za základ poměru se bere čas T metody RADIXSORT se základem 10 a počtem číslic POCCIF=4. Pole má 1024 prvků o délce jednoho prvku k=4 byty.

$\frac{T_i}{T}$	Poměr odvozený pro hypotetický počítač	Poměr výsledků z EC 1033
BUBBLESORT	64.0	68.5
BINARYINSERT	25.7	25.9
HEAPSORT	1.7	1.4

Tab. 7.6. Poměr časů pro teoreticky a experimentálně získané výsledky

7.10. Principy vnějších sekvenčních řadicích metod

Z předcházejících odstavců vyplynulo, že základním principem řadicích metod na vnějších, sekvenčně organizovaných paměťových médiích, je setřiďování (merging). Seznámíme se stručně se základními myšlenkami několika metod. Pro další úvahy budeme předpokládat, že seřazovanou datovou strukturou je soubor.

7.10.1. Metoda přímého setřiďování (straight merging)

Tato metoda potřebuje kromě souboru, v němž jsou seřazovaná data, dva další pomocné soubory.

V prvním kroku metody se data původního (zdrojového) souboru rozdělí rovnoměrně do dvou pomocných souborů. Ve druhém kroku se přečte z každého pomocného souboru jeden prvek, vytvoří se uspořádaná dvojice, která se uloží do zdrojového souboru. Tato akce se opakuje až se ze všech údajů vytvoří soubor uspořádaných dvojic. Ve třetím kroku se uspořádané dvojice rozdělí rovnoměrně do dvou pomocných souborů. Ve čtvrtém kroku se setříděním ze dvou souborů seřazených dvojic vytvoří soubor seřazených čtveřic atd. Tento cyklus se opakuje pro všechny 2^k-tice až po k, pro něž platí, že $2^{k+1} \geq N$. Jako poslední setřídíme poslední 2^k-tici a zbývající seřazenou posloupnost ve druhém souboru. Protože typickou implementací vnějšího sekvenčního souboru je magnetická páska, říká se této metodě také přímá metoda tří pásek. Algoritmus řazení můžeme v prvním přiblížení zapsat takto :

```
Rozděl prvky souboru f1 do pomocných souborů f2 a f3;
n:=0;
repeat l:=0;
    setříděním posloupností o délce 2^n z pásek f2 a f3
    vytvoř posloupnosti o délce 2^(n+1) a ulož je do souboru f1.
    S každou vytvořenou posloupností zvyš l o 1.
    if l>1 then
        Rozděl posloupnosti o délce 2^(n+1) ze souboru f1
        do souborů f2 a f3 ;
    n:=n+1
until l = 1;
```

Jednoduchou variantou této metody je přímá metoda čtyř pásek, která za cenu dalšího pomocného souboru (pásky) zkracuje proces vyloučením etapy rozdělení tím, že setřídění 2^{n-tice} se střídavě ukládají do jednoho ze dvou "cílových" souborů. Jakmile se "zdrojové" soubory vyčerpají, provede se záměna funkcí dvojic souborů. "Cílové" pásky se stanou "zdrojovými" a "zdrojové" se stanou "cílovými". Této metodě se říká také "přímé vyvážené setřiďování" (straight balanced merging).

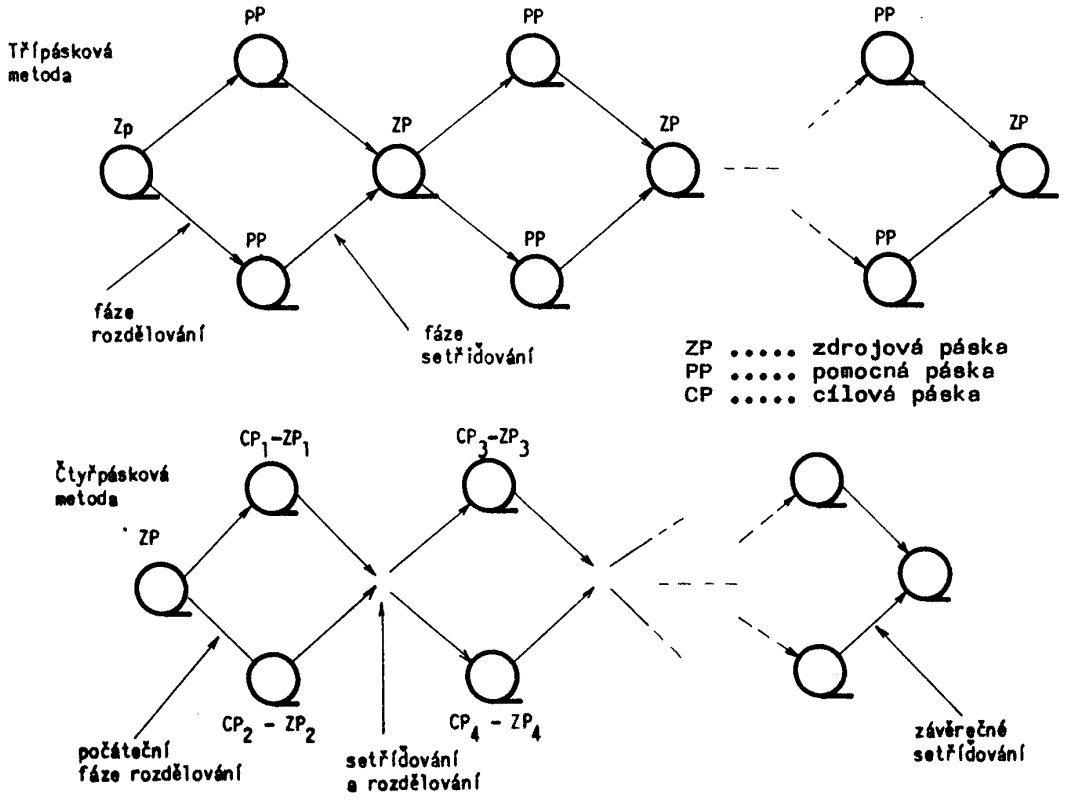
7.10.2. Metoda přirozeného setřiďování (Natural merging)

Na rozdíl od předchozí přímé metody tří pásek, přirozená metoda tří pásek nevytváří setříděné posloupnosti o délce 2^n při vzrůstajícím n, ale využívá počátečního uspořádání seřazovaného souboru. V první fázi rozdělování zapisuje střídavě do dvou pomocných souborů celé neklesající posloupnosti. Dále se již střídá fáze setřiďování a rozdělování s tím, že při setřiďování musí algoritmus rozeznat konec neklesající posloupnosti. Proces končí setříděním posledních dvou posloupností do zdrojového souboru.

Přirozená metoda čtyř pásek (také přirozené vyvážené setřiďování, angl. natural balanced merging) podobně jako u přímého setřiďování, zkracuje proces řazení vyloučením fází rozdělení za cenu použití dalšího pomocného souboru (pásky).

Schema postupu tří a čtyřpáskových metod je uveden na obr. 7.13.

Nelze-li u čtyřpáskové metody umístit v závěrečném setřiďování výsledný seřazený soubor na původní zdrojovou pásku, vloží se do procesu řazení kopírovací cyklus, který umístí seřazený soubor na původní pásku.



Obr. 7.13. Schema tří a čtyřpáskových metod

7.10.3. Mnohacestné vyvážené setřiďování (Balanced Multiway Merging)

Doba potřebná k seřazení sekvenčního souboru je úměrná počtu potřebných fází, kde každá fáze obsahuje průchod všemi daty seřazovaného souboru. Jedním ze způsobů, jak snížit počet průchodů spočívá v rozdělení posloupností do více než dvou souborů s následovným setřiďováním více zdrojových souborů do cílových souborů. Obsahuje-li seřazovaný soubor r neklesajících posloupností, pak jejich rovnoměrné rozdělení do N souborů a následné setřídění vytvoří r/N posloupností. Ve druhé fázi vznikne r/N^2 a k-té fázi r/N^k posloupností. Celkový počet přepisovacích operací pro N-cestné setřiďování (vyžadující 2N souborů) je v nejhorším případě dán vztahem

$$M = n \lceil \log_N n \rceil$$

kde n je počet prvků seřazovaného souboru.

Na rozdíl od dvoucestného vyváženého setřiďování (přirozená metoda 4 pásek) mnohacestné setřiďování končí v jedné fázi,až když jsou vyčerpány neklesající posloupnosti všech souborů.

Nechť jsou definovány typy TYPPOLOZKY a TYPPASKY. Definujme typ TYPCISLOPASKY = 1..P (kde P je sudý počet všech souborů - pásek), proměnnou fØ:TYPPASKY, která na počátku obsahuje seřazovaný soubor a proměnnou f:array [TYPCISLOPASKY] of TYPPASKY, která obsahuje používané soubory (pásky).

Aby byl proces záměny funkce zdrojových a cílových pásek jednoduchý, budeme místo přímého indexování souborů používat mapovací funkce implementované polem t:array [TYPCISLOPASKY] of TYPCISLOPASKY, a místo f[i] budeme používat zápis f[t[i]]. Počáteční stav mapovacího pole bude t[i]= i (pro i=1,...,P) a při výměně funkcí se vymění hodnoty pole t takto :

```
t[1]:=: t[P div 2+1]
t[2]:=: t[P div 2+2]
   ⋮
t[P div 2] :=: t[P]
```

a soubory t[1]až t[P div 2] budou vždy zdrojové a t[P div 2+1] až t[P] budou vždy cílové soubory.

Při prvním přiblížení bude mít program pro seřazení touto metodou tento tvar :

```
procedure MULTIMERGESORT;
  var i,j : TYPCISLOPASKY
        ℓ : integer; {počet rozdělovaných posloupností}
        t : array [TYPCISLOPASKY] of TYPCISLOPASKY;
begin {Počáteční rozdělení posloupností do pásek t [1]až t[n div 2]}
    j:=n div 2; ℓ:=Ø;
    repeat
        if j < n div 2 then j:=j+1
                       else j:=1;
      ┌──────────────────────────────────────────────┐
      │ Přepiš jednu neklesající posloupnost z pásky fØ na pásku j; │
      └──────────────────────────────────────────────┘
        ℓ := ℓ+1
    until eof (fØ);
    for i:=1 to n do t[i]:=i; {Inicializace mapovacího pole}
```

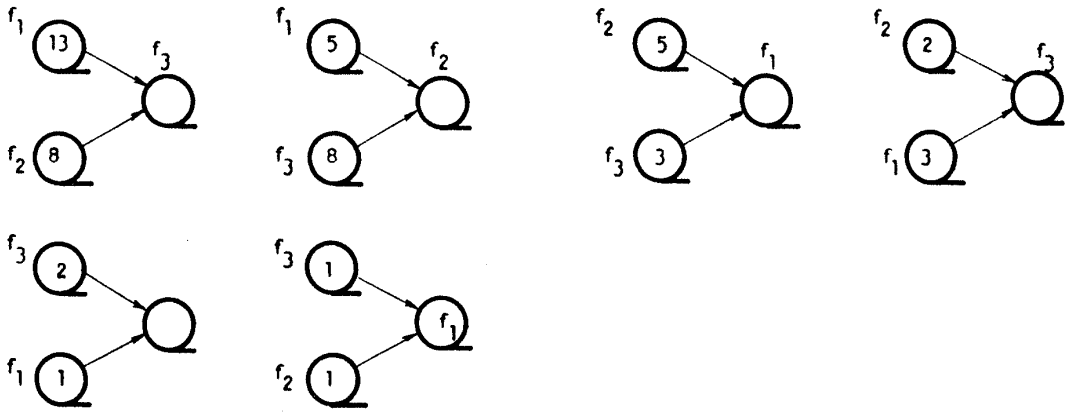
```
repeat {setřídění posloupností z t[1] až t[n div 2] do t[n div 2+1] až t[n]}
    Resetuj vstupní pásky;
      ℓ:=0;
      j:=n div 2+1;  {j je index výstupní pásky}
      repeat ℓ:=ℓ+1;
          setřiď jednu posloupnost ze vstupů do t[j]
          if j<n then j:=j+1 else j:=n div 2+1;
      until všechny vstupní pásky jsou vyčerpány;
    Zaměň funkci vstupních a výstupních pásek
until ℓ=1;
  {Páska t[1] obsahuje seřazený soubor}
end; {Procedury}
```

7.10.4. Princip polyfázového setřiďování (Polyphase Merge sort)

Vyvážení setřiďování zredukovalo opakování fáze rozdělení na jediné inicializační rozdělení a mnohacestné setřiďování významně urychlilo proces řazení za cenu zvýšení počtu potřebných souborů (pásek). Vzniká otázka, zda by vyššího počtu souborů nešlo využít účinněji. Princip polyfázového setřiďování si vysvětlíme na třípáskové variantě, která má stejné vlastnosti jako čtyřpáskové přirozené (vyvážené) setřiďování.

Předpokládejme, že posloupnosti souboru f_1 byly rozděleny inicializačním rozdělováním tak, že počty posloupností v souborech f_2 a f_3 jsou <u>různé</u>.

Posloupnosti ze zdrojových souborů f_2 a f_3 se setřiďují do cílového souboru f_1 tak dlouho, až ten ze zdrojových souborů, který obsahoval <u>méně</u> posloupností, vyčerpá. V tom okamžiku se proces setřiďování přeruší a dochází k <u>rekonfiguraci</u> funkce souborů. Vyčerpaný soubor se stane pro další proces <u>souborem cílovým</u> a bývalý cílový soubor se stane zdrojovým souborem. Proces seřazování pokračuje stejným způsobem až do okamžiku, kdy zbývají dva soubory, každý s jedinou posloupností. Tyto posloupnosti se setřídí do konečné seřazené posloupnosti. Postup znázorňuje obr. 7.14. Uvnitř symbolu souboru je uveden počet posloupností.



Obr. 7.14. Postup polyfázového setřídění pro 3 soubory

Postup můžeme znázornit tabulkou počtu posloupností v souborech, kde soubor s nulovým počtem posloupností je v dané fázi cílový a soubor s nejmenším počtem prvků (bude nejdříve vyčerpaný) se stane cílovým v další fázi. V tab. 7.7. a 7.8. jsou uvedeny stavy jednotlivých fází pro tří a šestipáskový systém při polyfázovém setřídění.

f_1	f_2	f_3	$\sum$
13	8	Ø	21
5	Ø	8	13
Ø	5	3	8
3	2	Ø	5
1	Ø	2	3
Ø	1	1	2
1	Ø	Ø	1

Tab. 7.7. Polyfázové setřiďování pro 3 soubory

f_1	f_2	f_3	f_4	f_5	f_6	$\sum$
16	15	14	12	8	Ø	65
8	7	6	4	Ø	8	33
4	3	2	Ø	4	4	17
2	1	Ø	2	2	2	9
1	Ø	1	1	1	1	5
Ø	1	Ø	Ø	Ø	Ø	1

Tab. 7.8. Polyfázové setřiďování pro 6 souborů

Jak určit inicializační rozdělení tak, aby se po některé fázi nestalo, že minimální počet je současně na více souborech? Z tab. 7.7. lze snadno vypozorovat, že počty posloupností v souborech vytvářejí v obráceném pořadí hodnoty Fibonacciho posloupnosti $F_{i+1}=F_i+F_{i-1}$ pro $F_0=Ø$ a $F_1=1$ (Fibonacciho posloupnost: 0,1,1,2,3,5,8,13,21,24,55,...). Z toho vyplývá, že pro třípáskový systém musí být při inicializačním rozdělování počty posloupností ve dvou souborech rovny dvěma po sobě jdoucím Fibonacciho číslům.

Zobecněním principu Fibonacciho posloupnosti získáme Fibonacciho posloupnost řádu p, která je definovaná takto :

$$F_{i+1}^{(p)} = F_i^{(p)} + F_{i-1}^{(p)} + \ldots + F_{i-p}^{(p)}$$

kde

$$F_p^{(p)} = 1 \quad \text{a} \quad F_i^{(p)} = Ø \quad \text{pro} \quad Ø \leq i < p$$

Až dosud používaná Fibonacciho posloupnost byla 1.řádu. Pro systém o 6-ti páskách budeme používat Fibonacciho čísel 4.řádu, kde

$$F_{i+1} = F_i + F_{i-1} + F_{i-2} + F_{i-3} + F_{i-4} \qquad (\text{pro } i \geq 4)$$
$$F_4 = 1$$
$$F_0 = F_1 = F_2 = F_3 = Ø$$

(Fibonacciho posloupnost 4.řádu má první členy : 0,0,0,0,1,1,2,4,8,16,31,61,120, 236,...)

Inicializačního rozdělení z 5-ti po sobě jdoucích Fibonacciho čísel (1,2,4,8,16) dostaneme takto :

f_1 bude mít 16 posloupností
f_2 - " - 16-1=15 posloupností
f_3 - " - 16-2=14 - " -
f_4 - " - 16-4=12 - " -
f_5 - " - 16-8=8 - " -

což předpokládá, že původní soubor měl právě 65 posloupností. Podobným způsobem z 5-ti po sobě jdoucích Fib.čísel (2,4,8,16,31) dostaneme inicializační rozdělení

(31,29,27,23,15) pro 125 posloupností. V tab. 7.9. je uveden počet posloupností umožňující dokonalé rozdělení ve 3 až 8 páskovém systému pro seřazení, ke kterému je zapotřebí p průchodů. Neobsahuje-li počáteční soubor počet posloupností, který se shoduje s některou sumou získanou výše uvedeným postupem, "doplníme" posloupnosti prázdnými posloupnostmi tak, aby celkový počet byl roven číslu umožňujícímu dokonalé rozdělení. Jinými slovy : známe přesně počet posloupností, které musí být pro dokonalé rozdělení v každém souboru. Má-li některý soubor ve skutečnosti méně posloupností nepovažujeme jeho vyprázdnění za ukončení aktivity. Prázdné posloupnosti ve skutečnosti nesetřiďujeme a výslednou prázdnou posloupnost "zapisujeme" do cílového souboru. Z tohoto důvodu je účelné rozdělit prázdné posloupnosti pokud možno rovnoměrně mezi n-1 souborů.

p \ n	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

Tab. 7.9. Počty posloupností pro dokonalé rozdělení ve 3 až 8 páskovém systému

7.11. L i t e r a t u r a

[1] Pačev,Č. : Efektivnost programů pro třídění ve vnitřní paměti
Diplomová práce na oboru Elektronické počítače
FE VUT v Brně, 1980.

[2] Knuth,D. : The art of computer programming
Vol. 3 Sorting and searching
Addison-Wesley, 1975

[3] Wirth,N. : Algoritmus + Data Structures = Programs
Prentice-Hall. INC. 1976.

[4] Wiedermann,J. : Triedenie
Sborník semináře SOFSEM'82, VUT UJEP Brno,
VVS Bratislava 1982.

[5] Revidovaná názvoslovná norma ČSN 36 9001 (ve zpracování)

[6] Colin Day, A. : Fortran techniques with special reference to non-numerical application. Cambridge University Press 1972

[7] Judd,D.R. : Použitie súborov ALFA Bratislava 1975

[8] Wiedermann,J. : Vyhľadávanie Sborník semináře SOFSEM'81, VUT UJEP Brno, VUS Bratislava, 1981

[9] Aho,A., Hopcroft,J., Ullman J. : Postrojenije i analiz vyčislitělnych algoritmov, Moskva, Mir, 1979