# NĚKTERÉ ALGORITMY ŘAZENÍ POLÍ

*Doc. Ing. Jan HONZÍK, CSc katedra počítačů FE VUT v Brně*
*Pavel KNOTEK, student 4. ročníku oboru Elektronické počítače FE VUT v Brně*

## 1. ÚVOD.

Příspěvek o řazení polí a souborů navazuje na serii příspěvků předcházejicich seminářů PROGRAMOVÁNÍ , které vycházely z temat obsažených v učebních osnovách předmětu Programovací techniky pro 3. ročník pětiletého studia oboru Elektronické počítače na elektrotechnických fakultách ČSSR.

V příspěvku je kromě základních pojmů a terminologie uveden výběr nejznámějších a nejvýznamějších algoritmů s vysvětlením jejich principu a ukázkou jejich implementace v jazyce PASCAL-ADT. Výsledky hodnocení metod vychází (na rozdíl od výsledků uvedených v [1]) z uvedené implementace a mají orientační charakter. Implementované programy vypracoval spoluautor referátu, student Pavel Knotek. V závěru referátu jsou uvedeny principy sekvenčního řazení.

## 2. ZÁKLADNÍ POJMY A TERMINOLOGIE

V praxi má pojem "třídění" tradiční význam, odvozený z dob mechanického řazení údajů na děrných štítcích, které se provádělo postupným tříděním štítků na třídicích strojích. V pozdější době byly na počítačích používány algoritmy , které s tříděním neměly nic společného ( výjimkou je "Radix-sort" viz.[1]), ale název spojený s tříděním jim v široké obci uživatelů již zůstal. Správné myšlení se však odráží jen ve správné terminologii a proto nemá smysl držet se tradice nepřesného pojmu.(Ostatně ani v tělocviku nevelíme:"Setřiďte se podle velikosti ...").

Podle názvoslovné normy ČSN [2] jsou pojmy z oblasti řazení definovány takto:

* *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 souborů 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.

V souvislosti s řazením uvedme některé další pojmy:

* *Sekvenčnost řazení* vyjadřuje, že algoritmus vystačí se sekvenčním přístupem k seřazovaným údajům a k meziproduktům řazení. Opakem je potřeba náhodného přístupu k seřazovaným údajům - tedy *nesekvenčnost*.
* *Prostorová a časová složitost* vyjadřuje míru prostoru resp. času, potřebnou k realizaci algoritmu. Používá se jí k oceňování různých metod řazení. O metodě, která dovede seřadit pole v jeho vlastním prostoru (bez potřeby dalšího významného prostoru) se říká že pracuje "*in situ*". O metodě, která k řazení již seřazené množiny potřebuje méně času než k seřazení množiny náhodně uspořádané a pro tu méně času, než k seřazení množiny opačně seřazené, se říká, že pracuje "*přirozeně*".
* *Stabilita* řazení je vlastnost algoritmu, který zachová relativní vzájemné pořadí položek se stejnými klíči. Tato vlastnost je významná pro postupné řazení podle více klíčů.

Na závěr tohoto odstavce si připomeňme, že základním smyslem řazení je *vyšší efektivnost vyhledávání.*

## 3. ŘAZENÍ PODLE VÍCE KLÍČŮ

Školním příkladem řazení podle více klíčů je řazení seznamu osob podle data narození, které sestává ze složek ROK, MESIC, DEN. Chceme-li vytvořit seznam osob podle stáří nebo seznam pořadí narozenin všech osob, nabízí se dva různé přístupy: a) *Postupným řazením podle jednotlivých klíčů se zvyšující se prioritou ( váhou ) klíčů* získáme seznam podle stáří tak, že řadíme pole (soubor ) postupně podle klíčů DEN, MESIC, ROK. Seznam narozenin získáme postupným řazením podle klíčů DEN, MESIC a chceme-li v seznamu dát přednost starším před mladšími, mají-li narozeniny ve stejný den, řadíme podle klíčů ROK, DEN, MESIC. Při postupném (vícefázovém) řazení podle více klíčů musí být zvolená metoda řazení *stabilní*. b) *Pro jednofázové řazení podle více klíčů* je nutné vytvořit *složenou relaci uspořádání* , která může mít tvar Booleovského výrazu nebo funkce . Pro jednodušší tvar seznamu narozenin by funkce měla tento tvar:

Nechť je dán typ

```
TYPDATNAR=record
              ROK:1900..2000;
              MESIC:1..12;
              DEN:1..31
          end;
function PRVNISTARSI(PRV,DRUH:TYPDATNAR):Boolean;
begin if PRV.MESIC<DRUH.MESIC
           then PRVNISTARSI:=true
           else if PRV.MESIC>DRUH.MESIC
                   then PRVNISTARSI:=false
                   else if PRV.DEN<DRUH.DEN {Měsíc je shodný}
                           then PRVNISTARSI:=true
                           else PRVNISTARSI:=false
end;{funkce}
```

Opakované vyčíslování relace může být časově náročné, a proto se hledají cesty ke zjednodušení . Jednou možností je *transformace* uspořádané n-tice klíčů do jedné hodnoty typu (nad nímž je v daném jazyce definována operace uspořádání), taková, aby relace uspořádání pro dvě různé n-tice zůstala po transformaci zachována. Příkladem takové transformace pro seznam narozenin je celočíselný výraz PORADI:=MES*31+DEN, kde hodnota proměnné PORADI může být klíčem pro jednofázové řazení.

V praxi se taková transformace provádí nejčastěji uspořádanou kumulací klíčů do paměťové oblasti , která může být chápána jako paměťová reprezentace typu, nad nímž je definována relace uspořádání.

Tato transformace je vysoce efektivní, je však pro netextové klíče *implementačně závislá* a předpokládá porušení kontroly typů (u jazyků, které ji disponují). Z toho důvodu je nutné znát zobrazení typu klíče v paměti daného počítače. Používá- li se např. doplňkový kód, nelze použít celočíselné klíče se zápornou hodnotou. Zcela problematické jsou klíče typu "real" , jsou-li zobrazeny jiným způsobem, než na počítačích JSEP a i tam lze použít jen kladných hodnot. Tento druh transformace je však výhodný pro textové klíče, a textový řetězec je vhodným cílovým typem transformace. Zkusme si cvičně upravit textový řetězec rodného čísla (RRMMDDPPPP) tak aby ve funkci klíče při řazení vytvářel oba uvedené seznamy a v případech shod dával přednost pánům před dámami (dle stáří) či dámám a starším (dle narození).

## 4. ŘAZENÍ BEZ PŘESUNU POLOŽEK

Nejvýznamějšími operacemi algoritmů řazení jsou porovnání klíčů dvou položek a přesun (resp. výměna) položek. Potřebný počet těchto operací vpodstatě určuje časovou složitost řadicí metody. Je-li přesouvaná položka paměťově rozsáhlá , pak je i její přesun časově náročný. Je-li k položkám náhodný přístup , lze řazení významně urychlit tak, že se nepřesouvají položky seřazovaného pole, ale pouze jejich "ukazatelé" v pomocném poli ukazatelů. Proces je znázorněn na obr.1.

POLE PRVNI 3

| | DATA | KLIC | POMPOLE | | POMPOLE | | POLERET |
|---|---|---|---|---|---|---|---|
| 1 | | 20 | 1 | | 3 | | 4 |
| 2 | | 40 | 2 | => | 1 | => | 0 |
| 3 | | 10 | 3 | | 4 | | 1 |
| 4 | | 30 | 4 | | 2 | | 2 |
| | | | po inicializaci | | po seřazení | | po zřetězení |

Obr.1. Řazení bez přesunu položek

Předpokládejme existenci datových typů a proměnných odpovídajících obr.1 a existenci inicializovaného pole POMPOLE. V původním řadicím algoritmu měla relace mezi prvky I a J tvar:

POLE[I].KLIC < POLE[J].KLIC

V algoritmu upraveném pro řazení bez přesunu položek bude mít tvar POLE[POMPOLE[I]].KLIC < POLE[POMPOLE[J]].KLIC

Odpovídající přesun (výměna) položek pak místo v poli POLE, proběhne v poli ukazatelů POMPOLE:

```
POM:=POMPOLE[I];
POMPOLE[I]:=POMPOLE[J];
POMPOLE[J]:=POM;
```

Pozn. Pro výměnu dvou prvků zavedeme pro zkrácení operaci ":=:" tedy např. POMPOLE[I]:=:POMPOLE[J] .

Výstupní seřazené pole lze z nezměněného zdrojového pole a z pole ukazatelů získat jedním průchodem:

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

Tato metoda vyžaduje dvojnásobek prostoru zdrojového pole. Je-li prostor limitujícím faktorem , lze situaci (za cenu zvýšení časové složitosti) řešit vytvořením *seřazeného zřetězeného seznamu* položek v poli POLE s pomocí dalšího pomocného pole ukazatelů POLERET a ukazatele PRVNI s následným seřazením na původním místě. Situaci znázorňuje nejpravější část obr.1., v němž hodnota 0 má vlastnosti pascalovského ukazatele *nil* a znamená, že prvek, jemuž tento ukazatel patří, je v seznamu poslední. Zřetězení provede úsek programu:

```
PRVNI:=POMPOLE[1];
for I:=1 to N-1 do POLERET[POMPOLE[I]]:=POMPOLE[I+1];
POLERET[POMPOLE[N]]:=0;
```

Algoritmus , který zřetězené pole seřadí "in situ", není vždy snadné napoprvé pochopit:

```
i:=1; POM:=PRVNI;
while i<N do begin
   while POM<i do POM:=POLERET[POM]; {Hledání následníka
                                      přesunutého na pozi-
                                      ci větší než i }

   POLE[I]:=:POLE[POM]; {Výměna prvků s indexy i a POM}
   POLERET[i]:=:POM;    {Výměna ukazatelů}
   i:=i+1    {Prvních i-1 prvků je již na svém místě}
end;
```

## 5. KLASIFIKACE PRINCIPŮ ŘAZENÍ POLÍ

Metody řazení polí lze podle principu, na kterém pracují, rozdělit do některé z těchto skupin:

a) Metody pracující na principu *výběru* (angl. selection) přesouvají postupně maximální (minimální) prvek ze seřazované množiny na konec výstupní seřazené lineární posloupnosti.

b) Metody pracující na principu *vkládání* (angl. insertion) zařazují další prvek "na řadě" do již seřazeného mezitvaru výstupní posloupnosti.

c) Metody pracující na principu *rozdělování* (angl. partition) rozdělují postupně všechny (pod)množiny na dvě nové podmnožiny tak, že všechny prvky jedné podmnožiny jsou menší než všechny prvky druhé podmnožiny. ( zobecněném přístupu jsou rozdělení i relace vícenásobné.)

d) Metody pracující na principu *sdružování* (nebo setřídění, angl. merging) sjednocují seřazené podmnožiny (na začátku třeba i jednoprvkové) postupně do větších seřazených podmnožin s cílem získat v konečné fázi celou množinu ve tvaru seřazené lineární posloupnosti.

e) Metody pracující na jiných principech nebo na základě kombinace výše uvedených principů.

## 6. ČASOVÁ SLOŽITOST ALGORITMŮ

Zdálo by se, že se zvyšující se rychlostí počítačů má úsilí o hledání rychlejších algoritmů řazení stále menší význam. Situace je však složitější. Předpokládejme, že k řešení daného problému je k dispozici 5 algoritmů s různou časovou složitostí, která je vyjádřena jako funkce počtu prvků (N). Tabulka tab.1. ukazuje, jak velký počet prvků (N) lze algoritmem řešit za předpokladu, že řešení se složitostí $S_1$=N pro 1000 prvků trvá 1s.

| Algoritmus | Složitost | Max. počet prvků (N) 1 s | 1 min | 1 hod | Vliv zrychlení 10x na původní rozsah $S_i$ |
|---|---|---|---|---|---|
| $A_1$ | $N$ | 1000 | $6x10^4$ | $3.6x10^6$ | $10xS_1$ |
| $A_2$ | $N \log_2 N$ | 140 | 4893 | $2x10^5$ | cca $10xS_2$ |
| $A_3$ | $N^2$ | 31 | 244 | 1897 | $3.16xS_3$ |
| $A_4$ | $N^3$ | 10 | 39 | 153 | $2.15xS_4$ |
| $A_5$ | $2^N$ | 9 | 15 | 21 | $S_5$+3.3 |

tab.1. Rozsah řešitelných problémů pro různé časové složitosti

Nejpravější sloupec ukazuje, jak se pro jednotlivé algoritmy zvětší počet prvků zpracovatelných za stejný čas, zrychlí-li se počítač 10x. Kdyby např. časová složitost algoritmů $A_1$ až $A_5$ měla tvar:

$CS_1=1000xN$

$CS_2=100xNxlog_2N$

$CS_3=10xN^2$

$CS_4=N^3$

$CS_5=2^N$

pak by jednotlivé algoritmy byly nejvýhodnější za těchto podmínek:

| | | |
|---|---|---|
| $A_5$ | pro | $2 <= N <= 9$ |
| $A_3$ | pro | $10 <= N <= 58$ |
| $A_2$ | pro | $59 <= N <= 1024$ |
| $A_1$ | pro | $N > 1024$ |

Z tohoto příkladu vyplývá, že nemusí platit, že nejrychlejší algoritmus (s lineární časovou složitostí) bude univerzálně nejvýhodnější, zejména vezmeme-li v úvahu, že často je rychlost algoritmu vykoupena náročností a složitostí logiky jeho řídicí a datové struktury.

V následujících odstavcích se budeme zabývat některými vybranými řadicími algoritmy s ohledem na optimální poměr jejich jednoduchosti a rychlosti a tudíž i použitelnosti v praxi. Nebudeme se zabývat analytickým vyjádřením jejich složitosti (je uvedeno v [1]), ale na závěr uvedeme tabulkové a grafické porovnání experimentálně zjištěných hodnot.

## 7. ŘAZENÍ NA PRINCIPU VÝBĚRU

Do této skupiny patří několik dobře známých algoritmů a řada jejich modifikací. Jednou z nejjednodušších je <u>select sort</u>. Je to <u>přirozeně</u> se chovající <u>nestabilní</u> metoda s kvadratickou časovou složitostí, vhodná pro N nepřevyšující pár desítek.

```
PROCEDURE SELECT_SORT(VAR POLE:TPOLE;N:INTEGER);
{*******************************************}
{*        RAZENI METODOU PRIMY VYBER       *}
{*******************************************}

VAR I,J,POZ:INTEGER;
BEGIN FOR I:=1 TO N-1 DO
      BEGIN POZ:=I;
            FOR J:=I+1 TO N DO
                IF POLE[J] < POLE[POZ] THEN POZ:=J;
            IF POZ<>I THEN  VYM(POLE[I],POLE[POZ]);
      END;
END;
```

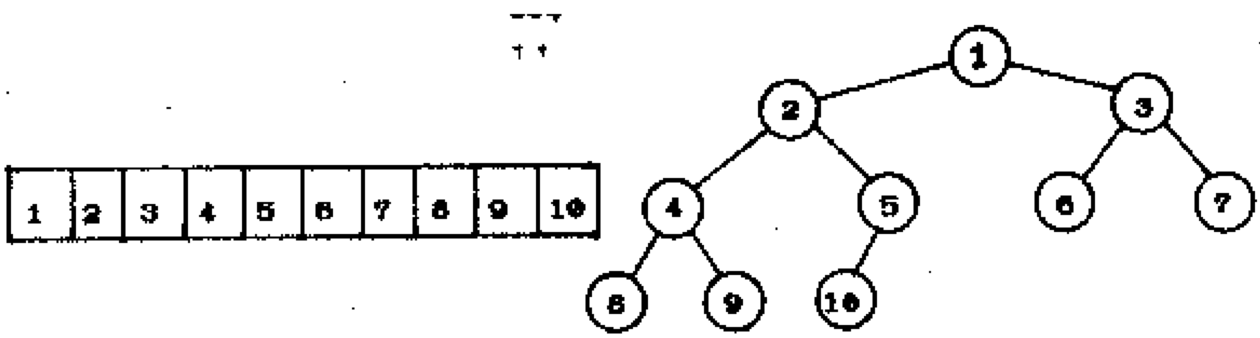
Velmi populární a notoricky nejhorší ze známých metod je metoda bublinového výběru - tzv. <u>Bubble-sort</u>. Je to <u>stabilní</u>

chovají symetricky z hlediska rychlosti s ohledem na smysl již seřazeného pole. Jinak lze tuto metodu i její varianty směle zařadit mezi metody vpodstatě *nevyhovující!*

```
PROCEDURE BUBBLE_B(VAR POLE:TPOLE;N:INTEGER);
{*********************************************}
{*  RAZENI METODOU BUBBLE SORT               *}
{*********************************************}

VAR I,KON,POZ:INTEGER;
BEGIN KON:=N - 1;
      REPEAT POZ:=1;
             I:=0;
             REPEAT I:=I+1;
                    IF POLE[I+1] < POLE[I] THEN
                    BEGIN POZ:=I;
                          VYM(POLE[I],POLE[I+1]);
                    END;
              UNTIL I = KON;
              KON:=POZ-1;
       UNTIL POZ = 1;
END;
$PAGE $
```

Nejvýznamnější metodou této skupiny je *řazení hromadou - "Heapsort"*. *Hromada* má strukturu binárního stromu, pro jehož každý uzel platí, že je větší (resp. menší), než oba jeho synovské uzly. Pak je v kořeni hromady *extrém* - tedy maximum (resp. minimum). Po přesunu extrému z kořene stromu do výstupní posloupnosti se na místo kořene přesune vhodný (nejnižší a nejpravější) uzel a pravidla hromady se znovuustaví *zatřesením* stromu (procedura SIFT), při němž nová hodnota kořene propadne na správné místo pro hromadu. Rychlost metody je dána rychlostí nalezení nového extrému. Po inicializačním ustavení hromady ze všech prvků, je délka "zatřesení" omezena průchodem od kořene k listu (tedy vytahem $\log_2 N$), kde N je postupně se snižující počet uzlů.

Binární strom hromady je sestaven s *"implicitním zřetězením"* t.zn. že uzel neobsahuje dva ukazatele na levý a pravý synovský uzel, nýbrž index synovských uzlů se určí z indexu otcovského uzlu. Je-li index otcovského uzlu i, pak index levého syna je 2*i a pravého 2*i+1. Z toho vyplývá, že každý vektor o N prvcích lze interpretovat jako binární strom, jehož kořenem je prvek na indexu 1 a listy stromu jsou prvky na indexech i, pro které platí: (2*i) > N. Situaci znázorňuje obr.2.



obr.2. Vztah binárního stromu a vektoru v hromadě Heap-sortu.

Algoritmus znovuustavení hromady lze slovně popsat takto:
Nechť je aktuálním uzlem kořen;
<u>repeat</u>
    Je-li aktuální uzel menší než větší ze dvou synovských uzlů
    (resp. menší než jediný - a to levý - synovský uzel),
    vyměň otcovský a vybraný synovský uzel mezi sebou,
    a aktuálním uzlem nechť je "propadlý" uzel
<u>until</u> (k výměně nedošlo) <u>or</u> (aktuální uzel je list);

Hromada se z neuspořádaného stromu (vektoru) vytvoří postupným "otřásáním" podstromů počínaje v kořeni rovném nejnižšímu a nejpravějšímu uzlu, který není listem. Jeho index je (N <u>div</u> 2) a na obr.2. je to uzel na i=5. Cyklus "otřásání" končí v kořeni a má tvar:

<u>for</u> i:=N <u>div</u> 2 <u>downto</u> 1 <u>do</u> *SIFT*(i,N);

Vlastní řazení je vlastně počítaným cyklem, ve kterém se první prvek (kořen- a maximum) vymění s aktuálně posledním prvkem a index aktuálně posledního se sníží o jedničku. Tím se kořen dostává do "zezadu" vytvářené seřazené posloupnosti a do kořene se dostává nejnižší nejpravější list.

```
PROCEDURE HEAP_SORT(VAR POLE:TPOLE;N:INTEGER);
(*********************************************)
(*  RAZENI S POUZITIM STROMOVE STRUKTURY    *)
(*********************************************)

VAR I:INTEGER;
    PROCEDURE SIFT(L,R:INTEGER);   [ "PROSETI" STROMU ]
    VAR J,X:INTEGER;JESTE:BOOLEAN;
    BEGIN J:=2*L;X:=POLE[L];
          JESTE:=J<=R;
          WHILE JESTE DO
          BEGIN IF J<R THEN
                      IF POLE[J]<POLE[J+1] THEN J:=J+1;
                      IF X > POLE[J] THEN JESTE:=FALSE
                      ELSE BEGIN
                           POLE[L]:=POLE[J];
                           L:=J;
                           J:=2*L;
                           JESTE:=J<=R;
                      END;
          END;
          POLE[L]:=X;
    END;
```

```
                                        { NA VRCHOLU SE OBJEVI MAXIMUM }
BEGIN                                   { VLASTNI SORT }
     FOR I:=N DIV 2 DOWNTO 1 DO SIFT(I,N);
     FOR I:=N DOWNTO 3 DO BEGIN VYM(POLE[1],POLE[I]);
                                        SIFT(1,I-1);
     END;VYM(POLE[1],POLE[2]);
END;
$PAGE $
```

Heap-sort je metoda, která se nechová přirozeně a je nestabilní. Pracuje ale "in situ" a je velmi rychlá při značné jednoduchosti struktury řízení. Pozornost si zasluhuje mechanismus hromady, umožňující rychlé postupné vyhledávání extrémů i její konstrukce vektorem.

## 6. ŘAZENÍ NA PRINCIPU VKLÁDÁNÍ

Řazení na principu vkládání se nejvíce podobá "ručním" metodám řazení a lze ho popsat takto:

```
for i:=2 to N do begin
    Najdi index K v intervalu <1..i-1>, na který se má
    zařadit prvek POLE[i], tak, aby úsek v daném intervalu
    zůstal seřazený;
    Posuň část pole od K do i-1 o jednu pozici doprava;
    Vlož do POLE[K] zařazovaný prvek
end;
```

Na tomto principu pracuje několik metod, z nichž zajímavé je "bublinové vkládání", které slučuje "hledání","vkládání" i posun pole do jednoho cyklu. Pro rozsáhlejší pole je nejzajímavější "vkládání s binárním vyhledáváním", které výrazně urychlí proces vyhledání místa pro vložení.

```
PROCEDURE BINARY_INSERT_SORT(VAR POLE:TPOLE;N:INTEGER);
{*************************************************************}
{*    RAZENI METODOU BINARNIHO PULENI INTERVALU             *}
{*************************************************************}

VAR I,K,L,R,POMO:INTEGER;
BEGIN FOR I:=2 TO N DO
        BEGIN L:=1;R:=I-1; POMO:=POLE[I];
              WHILE L <= R DO
              BEGIN M:=(L+R) DIV 2;
                    IF POMO < POLE[M] THEN R:=M-1
                                      ELSE L:=M+1;
              END;
              FOR K:=I-1 DOWNTO L DO POLE[K+1]:=POLE[K];
              POLE[L]:=POMO;
        END;
END;
$PAGE $
```

Zajímavou variantou je metoda pracující s dvojnásobně velkým polem, která po binárním vyhledání místa posouvá úsek ze dvou části pole rozdělených nalezeným indexem (levou doleva a pravou doprava). Tato varianta "ušetří" čas na přesunech za cenu dvojnásobného paměťového prostoru.

## 9. ŘAZENÍ NA *PRINCIPU* ROZDĚLOVÁNÍ – QUICK SORT

Quick-sort je jedním z nejznámějších algoritmů a je nejrychlejší z metod, jejichž strukrura je ještě dostatečně jednoduchá. Algoritmus je principiálně rekurzívní a jeho nerekurzívní zápis vyžaduje zásobník.

Základem algoritmu je uspořádání pole prvků do dvou částí, levé a pravé, takových, že prvky levé části jsou menší (nebo rovny) prvkům pravé části. Mechanismus takového uspořádání se pak aplikuje *rekurzivně znova* na vzniklou levou a pravou část tak dlouho, "pokud je co dělit".

Nejzajímavější částí je mechanismus rozdělování, jehož autorem je C.A.R.Hoare. Nejvýhodnější by bylo, kdyby rozdělením vznikly dvě stejně velké části. K tomu bychom ale potřebovali znát medián hodnot rozdělovaného intervalu (protože právě polovina všech hodnot je menší než medián). Algoritmus rozdělování pak prochází polem zleva a hledá první hodnotu větší (nebo rovnu) mediánu a pak zprava a hledá první hodnotu menší (nebo rovnu) mediánu. Tyto hodnoty mezi sebou vymění a průchod zleva a zprava (a tím i rozdělení) končí, když se sejdou (resp. "překříží") uprostřed. Najít medián by ale bylo pracné, a statisticky i experimentálně se prokázalo, že za "medián" lze prohlásit kterýkoli *prvek* intervalu. K tomu se dobře hodí prvek ze středu intervalu. Mechanismus rozdělení skončí nalezením indexů *I* a *J*, které se (I zleva, J zprava) právě překřížily, a jsou od sebe vzdáleny o 1 nebo o 2 (v druhém případě to znamená že prvek na indexu J+1 je právě roven "mediánu" a je již na "svém místě".

Pak vlastní algoritmus má tvar tří příkazů: rozdělení a dvou rekurzivních volání pro levou a pravou část rozděleného intervalu (pokud ovšem "je co" dělit).

```
PROCEDURE QUICK_SORT(VAR POLE:TPOLE;N:INTEGER);
$RECURSIVE ON$
   PROCEDURE Q_SORT(L,R:INTEGER);
   (**************************)
   (* REKURZIVNI CAST        *)
   (**************************)

   VAR I,J:INTEGER;
       PROCEDURE ROZDEL;  { ROZDELI POLE PODLE MEDIANU }
       VAR POMO:INTEGER;
       BEGIN I:=L;J:=R;
             POMO:=POLE[(I+J)DIV 2]; {MEDIAN }
             REPEAT WHILE POLE[I]<POMO DO I:=I+1;
                    WHILE POLE[J]>POMO DO J:=J-1;
                    IF I<=J THEN
                    BEGIN VYM(POLE[I],POLE[J]);
                          I:=I+1;J:=J-1;
                    END;
             UNTIL I>J;
       END;
   BEGIN ROZDEL;
         IF L<J THEN Q_SORT(L,J);
         IF I<R THEN Q_SORT(I,R);
   END;
$RECURSIVE OFF$
BEGIN Q_SORT(1,N);
END;
```

**Nerekurzívní zápis algoritmu využívá zásobníku k tomu, aby do něj uchoval jeden ze dvou podintervalů vzniklých dělením. Po rozdělení se tedy jedna část dále dělí a meze druhé se uloží na vrchol zásobníku. Není-li co dělit, vezme se pro další dělení interval, jehož meze jsou na vrcholu zásobníku - a dělí se dál. Je-li zásobník prázdný, algoritmus řazení končí. Zásobník musí pojmout tolik intervalů, kolik jich vznikne dělením. Proto je vtipnější dále dělit menší ze dvou intervalů a do zásobníku uložit meze většího. Pak vystačíme v nejhorším případě se zásobníkem pro $\log_2 N$ intervalů, což je pro běžné případy vždy menší než 20.**

```
PROCEDURE MODQUICK_SORT(VAR POLE:TPOLE;N:INTEGER);
(*********************************************)
(* NEREKURZIVNI VARIANTA QUICK SORTU          *)
(*********************************************)

CONST K = 13;
VAR I,J,L,R,X,S:INTEGER;
     STACK:ARRAY[1..K]OF RECORD LEFT,RIGHT:INTEGER;END;   { ZASOBNIK
   PROCEDURE PUSH(L,R:INTEGER);   { PUSH IN TO STACK }
   BEGIN STACK[S].LEFT  := L;
         STACK[S].RIGHT := R;
   END;
BEGIN S:=1 ; PUSH(1,N);     { INICIALIZACE ZASOBNIKU }
      REPEAT
         L:=STACK[S].LEFT ; R:=STACK[S].RIGHT;S:=S-1;
         REPEAT
            I:=L; J:=R ; X:=POLE[(I+J) DIV 2];        { MEDIAN }
            REPEAT
               WHILE POLE[I] < X DO I := I+1;
               WHILE POLE[J] > X DO J := J-1;
               IF I <= J THEN
               BEGIN VYM(POLE[I],POLE[J]);
                     I := I+1 ; J := J-1;
               END;

            UNTIL I > J;
            IF (J-L) < (R-I) THEN { DO ZASOBNIKU VZDY VETSI KUS }
            BEGIN IF I < R THEN BEGIN S := S+1 ; PUSH(I,R);END;
                  R := J;
            END
```

```
          ELSE BEGIN
                 IF J > L THEN BEGIN S := S+1 ; PUSH(L,J);END;
                 L := I;
          END;
        UNTIL L >= R;
      UNTIL S = 0;
END;
```

Quick-sort je metoda nestabilní. Je nejrychlejší (pro vysoká N je někdy rychlejší "Radix-sort" viz[1]) a měl by ho znát každý profesionální programátor. Jeho uvedená rekurzivní verze je potenciálním zdrojem větší spotřeby dynamicky přidělované paměti (nemá optimalizaci volby přednostního rekurzivního volání pro menší interval), což však většinou nevadí. Nerekurzivní verze je poněkud méně "čitelná", je však implementovatelná ve všech jazycích a lze ji v krajním případě optimalizovat na úrovni stroje s dobrou nadějí na další zrychlení.

## 10. ŘAZENÍ NA PRINCIPU SLUČOVÁNÍ - MERGE SORT

Merge Sort je výkonný, ale nikoliv jednoduchý a krátký algoritmus a pro nedostatek prostoru se spokojíme s principem: Algoritmus prochází zdrojovým polem z obou jeho konců a slučuje neklesající posloupnost zprava a zleva do jedné výsledné posloupnosti, kterou ukládá (zprava/zleva) do cílového (pomoc.) pole, které po prvním průchodu obsahuje jen polovinu původního počtu neklesajících posloupností. Poté se zamění funkce cílového azdrojového pole a proces se opakuje tak dlouho, až vznikne jedna neklesající posloupnost.

Tato metoda je nestabilní a potřebuje dvojnásobné pole a vzhledem k tomu, že algoritmus nepatří mezi nejjednodušší, není příliš atraktivní. Jeho výsledky jsou v tab.2. a na obr.3.

## 11. ŘAZENÍ SE SNIŽUJÍCÍM SE PŘÍRŮSTKEM - SHELL SORT

Shell Sort má mnoho variant, jež v historii řazení nabývaly větší či menší pozornosti. Využívá některé jednoduché přirozené metody řazení jako základ, a je založen na myšlence, že řazení je rychlejší, blíží-li se prvek k místu, na které patří po větších krocích (přírůstcích). Za tím účelem se řadí v jedné etapě separátně každý soubor prvků, které jsou od sebe vzdáleny o daný přírůstek. Tento přírůstek se v dalších etapách postupně snižuje až k hodnotě 1. Paradoxně působí skutečnost, že poslední etapa (s přírůstkem 1) by pole seřadila sama, bez předchozích etap. Protože však základem je přirozená metoda, působí řada předetap jako předtřiďovací a akcelerační nástroj pro poslední etapu. Jednotlivé varianty metody se od sebe liší především snižující se posloupností přírůstků.

Nyní již 3 desetiletí starý princip Shell Sortu ztratil význam v době, kdy se objevily modernější metody jako Heap Sort a Quick Sort, které byly rychlejší. V [3] se však objevila nová varianta, která je pozoruhodná svou <u>jednoduchostí</u> a <u>vysokou rychlostí</u>, kterou se řadí mezi Heap Sort a Quick Sort. Existují verze složitější a rychlejší než Heap Sort, ale žádná nepředčí Quick Sort. Zdá se, že tuto uvedenou metodu lze pro optimální kombinaci <u>jednoduchosti</u>, <u>stručnosti</u> a <u>rychlosti</u> korunovat za současného krále řadicích metod. Je dost stručná na to, aby se ji začátečníci a amatéři učili třeba nazpaměť, jako básničku mladých programátorů.

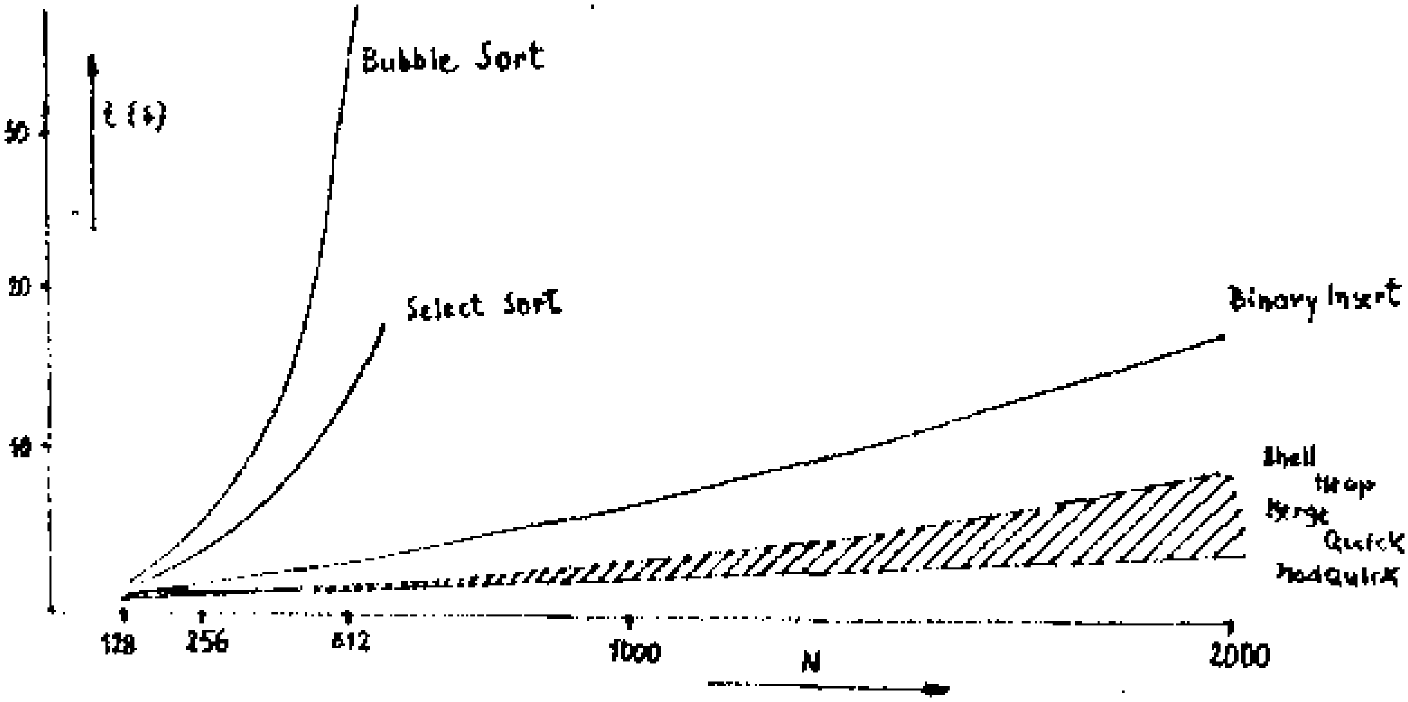
```
PROCEDURE SHELL_C(VAR POLE:TPOLE;N:INTEGER);
{*******************************************}
{*  RAZENI METODOU SHELL SORT              *}
{*******************************************}

VAR GAP,I,J:INTEGER;
BEGIN GAP:=N DIV 2;     { USTAVENI ZAKLADNIHO KROKU }
      WHILE GAP > 0 DO
      BEGIN FOR I:=GAP TO N-1 DO
            BEGIN J:=I-GAP+1;
                  WHILE (J>=1) AND (POLE[J]>POLE[J+GAP]) DO     {RAZENI}
                  BEGIN VYM(POLE[J],POLE[J+GAP]);
                        J:=J-GAP;
                  END;
            END;
     GAP:=GAP DIV 2 ;
     END;
END;
```

## 12. HODNOCENÍ A ZÁVĚR

V tab 2. jsou uvedeny délky řazení pro uvedené metody a pro různě dlouhá pole, zpracované na počítači ADT 4500. Časové údaje (s) je nutno brát orientačně. Z grafického vyjádření v obr.3. je patrné, že Quick Sort je absolutním vítězem uvedených metod. (Nutno podotknout, že Radix Sort s lineární časovou složitostí nebyl hodnocen; je však přeci jen složitější a další). Uvedená varianta Shell Sortu je však pro svou jednoduchost (nepotřebuje rekurzi ani zásobník) a stále značně vysokou rychlost metodou nejvýhodnější pro "masové nasazení".

| Metoda/délka | 128 | 256 | 512 | 1000 | 2000 |
|---|---|---|---|---|---|
| Select Sort | 0.98 | 3.80 | 14.73 | - | - |
| Bubble S | 1.24 | 4.72 | 30.69 | - | - |
| Heap Sort | 0.32 | 0.72 | 1.66 | 3.61 | 7.97 |
| Binary Insert | 0.31 | 0.93 | 3.04 | - | - |
| Shell S | 0.30 | 0.88 | 2.99 | 3.82 | 8.82 |
| Quick Sort | 0.28 | 0.58 | 1.25 | 2.60 | 5.47 |
| ModQuick Sort | 0.23 | 0.47 | 1.04 | 2.21 | 4.67 |
| Merge Sort | 0.31 | 0.66 | 1.34 | 2.97 | 6.52 |

tab.2. Délka řazení náhodně uspořádaného pole




obr.3. Grafické vyjádření hodnot z tab.2.

## 13. LITERATURA

[1] Honzík a kolektiv: Programovací techniky
skripta VUT Brno, 1984

[2] Revidovaná názvoslovná norma
ČSN 36 9001 - Počítače a systémy zpracování údajů,
Názvosloví (v tisku)

[3] Kernighan, B.W., Plaugher, P.J.: Software Tools in Pascal
Addison-Wesley, 1981