VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

Faculty of mechanical engineering
Institute of Automation and Computer Science

ALGORITMY TŘÍDĚNÍ
SORTING ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR THESIS

AUTOR PRÁCE **Jan Richter**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. Ing. Miloš Šeda, Ph.D.**
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2007/08

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Richter Jan

který/která studuje v **bakalářském studijním programu**

obor: **Strojní inženýrství (2301R016)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Algoritmy třídění

v anglickém jazyce:

Sorting Algorithms

Stručná charakteristika problematiky úkolu:

Algoritmy třídění (řazení) podle hodnoty klíčové položky jsou základní úlohou ve zpracování informací a lze je rozdělit na algoritmy třídění v operační paměti a na vnějších médiích.

Cíle bakalářské práce:

Cílem je popsat základní algoritmy pro třídění v operační paměti (Bubblesort, Quicksort, Heapsort, Mergesort, ...) a porovnat jejich časovou složitost.

Seznam odborné literatury:

Wirth, N.: Algoritmy a štruktúry údajov. Alfa, Bratislava, 1987.

Vedoucí bakalářské práce: doc. RNDr. Ing. Miloš Šeda, Ph.D.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2007/08.

V Brně, dne 5.11.2007

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ DĚKANÁT

doc. RNDr. Ing. Miloš Šeda, Ph.D.
Ředitel ústavu

v.z. R.

doc. RNDr. Miroslav Doupovec, CSc.
Děkan fakulty

Abstrakt

Bakalářská práce pojednává o základních typech třídicích algoritmů. Algoritmy lze rozdělit do dvou skupin - na jednoduché (přímé) a na složitější (optimalizované). Z přímých algoritmů třídění je práce věnována algoritmům přímé vkládání, přímý výběr a přímá výměna, kterou lze ještě rozdělit na třídění bublinové a třídění přetřásáním. Z optimalizovaných algoritmů je v práci popsáno třídění se zmenšováním kroku (Shellovo třídění) a třídění rozdělováním (Quicksort).

U zmíněných algoritmů se zabývám syntaxemi a také efektivností, s jakou jsou schopny utřídit danou množinu. Měřítkem efektivnosti jsou počty porovnávání a přesunů, které jsou funkcemi jediné proměnné, a to počtu prvků neuspořádané množiny.

Abstract

The bachelor thesis deals with basic types of sorting algorithms. We are able to divide these algorithms into two groups, which are elementary (direct) sorting algorithms and difficult (optimized) ones. From the group of direct algorithms, the thesis is dedicated to algorithms of direct paste, direct selection, and direct exchange, which could be further divided into bubblesort and shakersort. From the second group of optimized algorithms in the thesis, sorting with miniaturizing of steps (Shellsort) and dividing sorting (Quicksort) are described.

In context of these algorithms, I deal with syntaxes and also effectiveness, with which they are able to sort the given system. Criteria of effectiveness are comparison and movement calculations, which are the function of only one parameter – the number of members in the system.

Klíčová slova:

algoritmus, přímé vkládání, přímý výběr, přímá výměna, bublinové třídění, třídění přetřásáním, třídění rozdělováním, třídění zmenšováním kroku, efektivnost, porovnání, přesun.

Keywords:

algorithm, direct paste, direct selection, direct exchange, bubblesort, shakersort, quicksort, Shellsort, effectiveness, comparison, movement.

Poděkování:

Na tomto místě bych chtěl poděkovat svému vedoucímu doc. RNDr. Ing. Miloši Šedovi, Ph.D. za ochotu, cenné rady a připomínky, které mi byly při zpracovávání bakalářské práce velkou oporou.

Prohlášení:

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, atd.) uvedené v přiloženém seznamu. Některé názvy uvedené v této práci mohou být registrovanými značkami.

Z mé strany nejsou námitky proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Brně dne 23. května 2008 ……

Jan Richter

Obsah:

1. Úvod:

S tříděním se každý z nás setkává dnes a denně a mnohdy si to ani neuvědomuje. Jde o proces, v němž uspořádáváme rozházené prvky tak, jak bychom potřebovali. Můžeme uspořádávat karty, knihy, kompaktní disky, prádlo do zásuvek, jídlo do ledničky... Uspořádaná množina má potom předepsaný řád. Ať už proto, abychom se v ní lépe orientovali, například abychom snáze našli potřebné ponožky, nebo z jiných důvodů, aby se jídlo nezkazilo či aby se vůbec vešlo do ledničky.

To vše jsou různé procesy třídění, avšak jejich algoritmus se může dosti různit. Můžeme například z nákupu nejprve vybrat všechna mražená jídla a umístit je do mrazáku, potom všechna, co patří do ledničky a nakonec ta, která nijak chladit nepotřebují. Můžeme však také brát jídla po jednom tak, jak jsou umístěna v nákupní tašce a běhat s každým na potřebné místo, ať už k mrazáku, k ledničce nebo pouze k příslušnému regálu.

Způsobů, jak třídit objekty, je velké množství. V této práci bych chtěl prezentovat a popsat základní takové procesy.

„Tříděním obecně rozumíme proces přeuspořádání dané množiny objektů ve specifickém pořadí. Účelem třídění je ulehčit pozdější vyhledávání prvků tříděné množiny.“ [1 - citace ze str. 90]

Třídit můžeme buď podle velikosti, např. od nejmenšího po největší, tedy po setřídění bude pro n-té číslo platit, že je větší, než číslo na pozici n–1, a menší, než číslo na pozici n+1, nebo podle klíče. Klíč je převod, kterým převedeme jednotlivé prvky množiny na čísla, podle nichž třídíme. Např. můžeme chtít utřídit prvky –2; 4; –5; 1; –6

a) podle velikosti: –6; –5; –2; 1; 4

b) podle velikosti absolutní hodnoty: 1; –2; 4; –5; –6

Klíčem je v prvním případě pouhá hodnota čísla, v druhém je to absolutní hodnota. Klíčem však může být i jakákoliv funkce a například třídění podle abecedy představuje přiřazování číselných hodnot jednotlivým písmenům a jejich následnému porovnávání. Klíčem je pak právě toto přiřazení: tj. „A“ := 1; „B“ := 2 atd.

V práci se budu zabývat jednotlivými metodami třídění, jejich výhodami, nevýhodami, vhodností použití a sestavením algoritmů pro tyto procedury. Pro ilustraci použiji program Pascal, protože je to jednoduchý a pro problém plně dostačující programovací jazyk.

Programy jsou psané v knižní formě. Nelze je doslova opsat do programu Pascal. Obsahují např. vysvětlující komentáře k některým dílčím operacím a také obecnější příkazy, se kterými program Pascal neoperuje (např. „*x*.klíč“), které však budou v práci vysvětleny.

2. Typy údajů

V matematice je třeba rozdělit proměnné podle určitých význačných vlastností. Je třeba odlišit reálné proměnné od těch komplexních či logických. Proměnné také mohou být jednoduché či nabývat množiny hodnot nebo dokonce množiny množin.

V programovacím jazyku Pascal se proměnné definují pomocí příkazu **var** ještě před vlastním tělem programu. Za tímto příkazem nastává deklarace. Například kdybychom chtěli proměnné *a* a *b* nastavit jako celá čísla, docílíme toho příkazem *a*, *b*: integer; , kde integer je název právě pro množinu celých čísel.

Jaké typy údajů budeme používat, se zmíním v následujících kapitolách.

2.1 Standardní primitivní typy údajů [2]

Jsou to takové typy údajů, které není třeba nijak definovat a programovací jazyk je s nimi standardně obeznámen. V našem programu je tedy nemusíme nijak definovat. Jsou již připraveny k použití.

Ke konstrukci všech algoritmů třídění, které tato práce obsahuje, jsem potřeboval tyto následující standardní primitivní typy údajů:

- **integer**

Proměnná typu integer je celočíselná proměnná. Množina celých čísel je uzavřená ke sčítání, odčítání a násobení. Proměnné typu integer je tedy vhodné použít k indexování či jako počítadla operací.

- **boolean**

Proměnná typu boolean je logická proměnná. Může tedy nabývat pouze dvou hodnot, a to pravda (1) či nepravda (0). V programovacím jazyku Pascal je pravda psaná jako „true“, nepravda jako „false“. Proměnné typu boolean se používají k zastavování cyklů a vyskytují se v podmínkách. Sdělení, že logická proměnná *a* neplatí, můžeme napsat jako „*a* = false“ nebo také „**not** *a*“.

- **real**

Proměnná typu real je proměnná nabývající hodnot reálného čísla. Množina reálných čísel je uzavřená ke sčítání, odčítání, násobení, dělení a odmocnině z nezáporného čísla. Proměnné typu real se používají pro všechny výše popsané operace. Nehodí se však k indexování.

2.2 Primitivní typy údajů [3]

Obecné primitivní typy údajů programovací jazyk nezná. Je třeba je tedy nastavit, a to ještě před tím, než dochází k deklaraci proměnných, protože právě tyto typy se v deklaracích používají. Činí se tak po příkazu **type**.

V našich programech budu používat jen jeden pro Pascal neznámý typ proměnné, a tím je prvek.

Typ prvek však nemohu blíže specifikovat, protože je odvozen z toho, jaký druh prvků budeme mít za úkol setřídit. Pokud budeme třídit podle abecedy, je typ prvek roven typu char (=znak). Pokud budeme třídit určitá čísla, bude typ prvek roven typu integer, popř. real. Lze však třídit i prvky úplně jiného charakteru, jak už jsem popsal výše. Omezíme se tedy na to, že typ prvek je odvozen ze vstupních prvků. Proto má typ také tak příznačný název.

2.3 Struktura pole [3]

V našich programech budeme používat rovněž proměnné typu pole. Pole o rozsahu 1-*n* se používá tehdy, když máme *n* rovnocenných vstupních hodnot, se kterými se budou provádět tytéž operace. Např. deklarace proměnné „*a*: **array** [1..50]“ **of** integer znamená, že budeme moci použít až 50 rovnocenných proměnných typu integer, které budou indexovány. Vzniknou tedy proměnné a_1, a_2, a_3, … a_n. Do proměnné typu pole budeme ukládat všechny vstupní hodnoty nesetříděné množiny a rovněž výstupní setříděné hodnoty.

Snahou je použít pro vstupní i výstupní množiny pouze jednu strukturovanou proměnnou typu prvek, v rámci které se během třídění budou jednotlivé prvky přesouvat a prohazovat. K tomu budeme nutně potřebovat ještě pomocnou proměnnou rovněž typu prvek, jinak bychom nezabránili přepsání jedné či více vstupních proměnných. Pokud máme přesunout prvek s indexem *i* na místo prvku *j*, mohli bychom si prvek *j* přepsat, a tudíž jej musíme před tímto přesunem uložit právě do pomocné proměnné *x*.

3. Základní typy třídících algoritmů

V následující kapitole se budeme zabývat už konkrétními třídicími algoritmy.

Naše nesetříděná množina je tvořená strukturovaným polem a její *i*-tý prvek nese označení *a*[*i*]. Prvky pole jsou typu prvek. Počet prvků množiny označíme *n*.

Již bylo zmíněno, že nemusí být patrné, jak mají být prvky seřazeny. Pokud jsou prvky písmena a řadíme je podle abecedy, čísla a řadíme podle funkčních hodnot, nebo pokud chceme aplikovat úplně jiný způsob řadění, je třeba prvky typu prvek převést na číselnou podobu, v níž už prvky lze správně porovnat. Tento převod nazýváme klíčem. Je třeba si uvědomit, že v následujících programech nebudeme porovnávat přímo prvky, ale jejich klíče.

Označením *x*.klíč nebo *a*[*i*].klíč myslíme převod proměnných typu prvek na proměnné, jejichž hodnoty mezi sebou lze porovnat. Např. mějme prvky *a*[1]=‘a‘ a *a*[2]=‘b‘. Zápis *a*[1] <> *a*[2] nemá smysl, protože nelze rozhodnout, jestli je ’a‘ větší nebo menší než ‘b‘. Klíč v tomto případě znamená převod znaků na čísla patrně tak, jak jdou podle abecedy. Prvek *a*[1] má hodnotu „a“ a jeho klíč bude mít hodnotu 1. Zápis *a*[1].klíč <> *a*[2].klíč už tedy smysl má, protože jeho význam je 1<>2, což už lze rozhodnout.

3.1 Třídění s přímým vkládáním

Metoda přímého vkládání je specifická například pro hráče karet. [1] Karty v ruce hráče rozdělíme do dvou skupin - na karty utříděné a na karty neutříděné. Neutříděné karty pak po jedné bereme a hledáme v množině utříděných karet správné místo, kam ji zařadit. Množina utříděných karet se nám tedy o jeden prvek zvětšila, ale utříděná zůstává. Takto pokračujeme, dokud v množině neutříděných karet nezbývá žádná karta, tedy všechny karty jsou utříděné.

Př.1: Příklad hledání vhodného místa:

	Utříděná množina	Neutříděná množina
Začátek	2; 3; 6; 10; 12	5; 8; 4; 11
1. krok	2; 3; 5; 6; 10; 12	8; 4; 11
2. krok	2; 3; 5; 6; 8; 10; 12	4; 11
3. krok	2; 3; 4; 5; 6; 8; 10; 12	11
4. krok	2; 3; 4; 5; 6; 8; 10; 11; 12	x

Vidíme, že v každém kroku se utříděná množina o jeden prvek zvětší a neutříděná o jeden zmenší. Pokud začínáme soubor teprve třídit, je utříděná množina jednoprvková a představuje ji prvek s indexem 1. Neutříděnou množinu tvoří prvky 2, 3…*n*, kde *n* je celkový počet prvků, které chceme setřídit.

Postup při hledání vhodného místa:
Pro ilustraci se věnujme 1. kroku v Př.1.

1) Vezmeme první prvek neutříděné množiny (*a*[6]), vložíme jej do proměnné *x*. V našem případě to bude prvek 5.
2) Víme, že množinu máme třídit podle velikosti od nejmenšího po největší. Tedy hledáme pozici *k*, pro kterou platí, že číslo na pozici *k*–1 je menší než 5 a číslo na pozici *k*+1 je větší než 5.

3) Prvek 5 je před celým krokem zařazený na pozici 6, tedy na první pozici, která nespadá do utříděné množiny. Abychom jej správně zařadili, vložíme jej do proměnné *x*, kterou pak postupně porovnáváme s jednotlivými prvky utříděné množiny. Nejprve porovnáme obsahy proměnných *x* a *a*[5] (poslední prvek utříděné množiny). Pokud by platilo *a*[5]<*x*, věděli bychom, že prvek x patří na konec utříděné množiny. Pokud ne, věděli bychom, že prvek *x* patří před prvek *a*[5] a porovnávali bychom ho postupně s dalšími prvky utříděné množiny, dokud bychom nenalezli prvek a[*i*], který by byl již menší než prvek *x*. Znamená to, že prvek *x* patří za prvek *a*[*i*], tedy po dokončení kroku bude uložen v proměnné *a*[*i*+1].
4) Prvek *x* však nemůžeme uložit do proměnné *a*[*i*+1], protože bychom si přepsali hodnotu, kterou tato proměnná aktuálně obsahuje. Víme ale, že proměnnou *x* přesouváme z pozice 6. Proto postupujeme tak, že do proměnné *a*[6] uložíme proměnnou *a*[5], do *a*[5] uložíme *a*[4]… a do proměnné *a*[*i*+1] uložíme nakonec hodnotu proměnné *x*.

Protože děje 3) a 4) probíhají se stejným načasováním, lze je zapsat do jednoho cyklu s jedinou řídicí proměnnou.
Celý krok 1 vypadá tedy schematicky následovně:

```
                    ↑5
2; 3; 6; 10; 12;…
                  5
2; 3;  6; 10;…;12;
              5
2; 3; 6;…; 10; 12;
           ↓5
2; 3;…6; 10; 12;
```

Počet kroků je roven počtu prvků v celkové množině –1 (první prvek je již utříděný).

Algoritmus třídění přímým vkládáním pak vypadá následovně [1] :

```
procedure trideniprimymvkladanim;
    var i,j: integer; x: prvek;
begin
  for i:= 2 to n do
    begin  x:= a[i];                  a[i] je první nesetříděný prvek
           a[0]:= x;                  tímto zabráníme tomu, aby cyklus pokračoval
                                      pod nulu. Pokud bude x.klíč nejmenší, cyklus
                                      skončí pro j=1
           j:= i–1;                   v tomto okamžiku je j rovno aktuálnímu počtu
                                      setříděných prvků
           while x.klíč < a[j].klíč do   hledání vhodného místa pro prvek x
                 begin  a[j+1]:= a[j];   přesouvání prvků o jednu pozici dál
                        j:= j–1
                 end;
           a[j+1]:=x                  a[j].klíč je než x.klíč, tedy  x patří na pozici j+1
    end
end
```

Efektivnost cyklu můžeme posuzovat z hlediska počtu porovnávání klíčů C a počtu přesunů M (přiřazení jedné hodnoty proměnné do jiné proměnné). Počet porovnávání klíčů je v každém kroku nejméně 1 a nejvíce i–1. Pokud jsou všechny permutace stejně pravděpodobné, průměrně bude počet porovnávání roven (1+i–1)/2 = i/2. Počet přesunů v kroku je $M_i = C_i + 1$ včetně zarážky (a[0]=x).

Celkový počet porovnávání a přesunů ve všech krocích pak je:

$C_{min} = n - 1$	$M_{min} = 2(n - 1)$
$C_{prům} = ¼(n^2 + n - 2)$	$M_{prům} = ¼(n^2 + 9n - 10)$
$C_{max} = ½(n^2 + n) - 1$	$M_{max} = ½(n^2 + 3n - 4)$

Hodnoty C a M se liší podle utříděnosti původní množiny. Pokud je původní množina již utříděná, v kroku se provede jen jediné porovnávání (x.klíč<>a[j].klíč) a dvě přiřazení (x:=a[i]; a[0]:=x;). Počet kroků je roven n–1, viz výše. Naopak při dokonale nesetříděné množině (tj. při množině setříděné v našem případě od největšího po nejmenší) musí cyklus v každém kroku vykonat všechna možná porovnání a přesuny.

Algoritmus přímého vkládání lze zlepšit převodem na tzv. *binární vkládání*. Jeho podstatou je rychlejší nalezení místa v utříděné množině. Posloupnost binárního vkládání toto místo hledá postupným celočíselným dělením utříděné množiny. Nezařazené číslo porovnáme nejprve s prostředním číslem utříděné množiny. Jestliže bude vyšší (nižší), budeme hledat vhodné místo dále jen v horní (dolní) polovině utříděné množiny a hledat ho budeme obdobným způsobem. Jde tedy o značné zkrácení cesty k nalezení správného místa. Prvky od tohoto místa dále je však třeba opět po jednom posunout, aby se uvolnilo místo pro nový prvek x.

Algoritmus třídění binárním vkládáním pak vypadá takto [1] :

```
procedure tridenibinarnimvkladanim;
    var i, j, l, r, m: integer; x: prvek;
begin
  for i:= 2 to n do
    begin  x:= a[i]
           l:= 1;                                  index první proměnné prohledávané množiny
           r:= i–1;                                index poslední proměnné prohledávané množiny
           while l<= r do
                 begin
                   m:=(l+r) div 2;
                   if x.klíč<a[m]  then r:= m–1    hledané místo je ve spodní polovině, jejíž nejvyšší proměnná má index m–1
                                   else l:= m+1    hledané místo je v horní polovině, jejíž nejnižší proměnná má index m+1
      end
      for j:= i–1 downto l do a[j+1]:= a[j];       posunutí hodnot od místa l o 1 dozadu
      a[l]:= x;                                    uložení hodnoty x na pozici l
    end
end
```

Pokud bude platit *a*[*j*]klíč<=*x*.klíč<=*a*[*j*+1].klíč, najde se místo pro uložení prvku. Zkoumaná utříděná množina se tak dlouho půlí, až nakonec zůstane jednoprvková. Interval složený z *i* klíčů se tedy půlí vždy $\log_2 i$ krát. Počet porovnávání potom bude:

$$C = \sum_{i=1}^{n} \log_2 i \quad [1]$$

Aproximací sumy pomocí integrálu dostáváme:

$$\int_1^n \log x dx = \left[x(\log x - c)\right]_1^n = n(\log n - c) + c \quad [1]$$

V posloupnosti se vyskytuje operátor celočíselného dělení. Například výrazu 7 div 2 náleží hodnota 3. Pokud je klíč zkoumaného prvku menší než klíč prvku 3, počítáme poté s intervalem 1 až 2, což je dvouprvkový interval. Pokud je *x*.klíč větší než *a*[3].klíč, počítáme s intervalem 4 až 7, což je interval tříprvkový.

Z toho můžeme usoudit, že posloupnost najde vhodné místo rychleji, pokud se toto nachází na počátku utříděné množiny, než kdyby bylo na konci. Skutečný počet porovnávání tak může být o 1 větší, než se očekávalo.

Pokud budou prvky v neuspořádané množině uspořádány naopak (v našem případě od největšího po nejmenší), proběhne třídění rychleji, než kdyby se měla přeuspořádat množina uspořádaná:

Např: mějme prvky 10, 9, 8, 7… V každém kroku se místo pro *x*.klíč nachází na počátku uspořádané množiny. Toto místo bude nalezeno vždy nejrychleji.

Kdyby byly prvky již uspořádány, tj. 7, 8, 9, 10… , místo pro prvek *x* by se nacházelo vždy na konci uspořádané množiny. Tedy toto místo se hledá vždy nejdéle.

Proto je algoritmus přímého vkládání vhodný na co nejvíce neuspořádané množiny, kdežto binární vkládání je vhodné pro téměř utříděné. Vylepšení přímého na binární vkládání tedy nemusí vždy přivést takový efekt, jako se očekávalo.

Třídění přímým i binárním vkládáním má však stejný charakter co se týče přesunů, které obecně vyžadují mnohem více času než prostá porovnávání. V obou případech je hodnota veličiny *M* (počet přesunů) = n^2. Zmenšení hodnoty veličiny *M* bychom mohli docílit tím, že se prvky budou přesouvat na delší vzdálenosti.

3.2 Třídění přímým výběrem

Metoda třídění *přímým výběrem* se vyznačuje přesuny prvků na větší vzdálenosti. Princip spočívá ve výběru prvku s nejnižším klíčem *a*[*k*] a v dosazení na první místo nesetříděné množiny. Abychom neztratili prvek s indexem 1, dosadíme ho zpět na místo *k*. Víme, že na pozici 1 je už definitivně prvek *a*[*k*], proto se dále zabýváme čísly od pozice 2 výš.

Příklad třídění přímým výběrem:

	4, 8, 5, 2, 6	
1. krok	**2**, 8, 5, **4**, 6	nejnižší prvek (2) se prohodil s prvkem na pozici 1(4)
2. krok	2, **4**, 5, **8**, 6	prvek dva zůstane už definitivně na pozici 1. Zabýváme se nyní pozicí 2, na kterou dosadíme nejnižší číslo zbylého intervalu (4).

3. krok	2, 4, **5**, 8, 6	nejnižší číslo je nyní na právě zkoumané pozici 3. K žádnému prohození nedojde. Ale je důležité, abychom na začátku každého kroku *i* přiřadili nejprve nejnižší hodnotu právě prvku *i*. Kdybychom to neudělali, na místo *i* by se mohl přesunout prvek z pozice, kterou si počítač z předešlého kroku pamatuje jako pozici nejmenšího čísla. V našem případě by se na místo 3 přesunul prvek z pozice 4, tedy prvek 8.
4. krok	2, 4, 5, **6**, **8**	

Je zřejmé, že počet kroků je roven *n*–1, kde *n* je počet prvků množiny. „–1“ je ve vzorečku proto, že v *n*-tém kroku by se prvek pro pozici *n* hledal z jednoprvkové množiny, což je zbytečné, protože po kroku *n*–1 už víme, že prvek *n* má největší klíč. V každém kroku *i* hledáme prvek na pozici *i*. Tento prvek bude mít nejmenší klíč z aktuálně prohledávané podmnožiny *i*-*n* a v celkové množině bude mít *i*-tý nejmenší klíč.

Postup při hledání prvku s nejnižším klíčem spočívá v procházení prohledávané nesetříděné množiny a porovnávání klíčů jednotlivých prvků s klíčem prvku, který si pamatujeme jako aktuálně nejmenší. Pokud najdeme prvek s ještě menším klíčem, stane se porovnávaným prvkem právě tento.

Např.: hledáme nejmenší prvek množiny

	5, 4, 8, 2	
1.	**5**, 4, 8, 2	nejmenší prvek je v 1. fázi aktuálně první prvek
2.	5, **4**, 8, 2	porovnáme aktuálně nejmenší prvek s druhým v pořadí, zjišťujeme, že prvek na pozici 2 je menší než prvek na pozici 1 a pozici dva si tedy zapamatujeme namísto pozice 1
3.	5, **4**, 8, 2	nic se nemění, protože prvek z pozice 3 je větší než aktuálně nejmenší prvek
4.	5, 4, 8, **2**	na pozici 4 je menší prvek, než náš aktuálně nejmenší

Z celého hledání tedy vyjde výsledek, že nejmenší číslo je na pozici 4.

Úplný algoritmus třídění přímým výběrem vypadá tedy takto [1] :

```
procedure trideniprimymvyberem;
    var i, j, k: integer; x: prvek;
begin for i:= 1 to n–1 do                hledáme vždy prvek s i-tým nejmenším klíčem
                                         pro pozici i
        begin k:= i; x:= a[i]            v prvním kroku uvažujeme za prvek s nejmenším
                                         klíčem prvek na pozici i. Viz výše
            for j:= i+1 to n do          prohledáváme interval od i+1 do n
                if a[j].klíč<x.klíč then pokud najedeme prvek s menším klíčem a[j] než
                    begin k:= j; x:= a[j]; má prvek x z pozice k s aktuálně nejmenším
                    end                  klíčem, prvek a[j] i pozici j si zapamatujeme
            a[k]:= a[i]; a[i]:= x        po nalezení prvku s nejmenším klíčem
        end                              z intervalu i-n dojde k prohození prvků i a k.
end
```

Počet porovnávání jednotlivých klíčů *C* je pro každý krok roven *n*–*i*, kde *n*-*i* je počet prvků, z nichž se hledá ten s nejmenším klíčem. Protože *i* jde od 1 do *n*, v průměru bude *i* rovno *n*/2. Tedy počet porovnávání je v průměru roven *n*–*n*/2 = *n*/2. Již dříve jsme

odvodili, že počet kroků je roven n–1. Tedy celkový počet porovnávání $C = n/2 \,.\, (n–1)$. Po jednoduché úpravě dostaneme $C = ½ \,(n^2 – n)$.

Po každém kroku se provedou nejméně tři přesuny, a to když je prvek s nejmenším klíčem aktuálně na pozici i. Počet kroků je roven n–1, a tedy minimální celkový počet přesunů $M_{min} = 3(n\text{-}1)$. Maximální možný počet přesunů bude u množiny uspořádané obráceně, $M_{max} = \text{trunc}\,(n^2/4) + 3(n – 1)$. [1]

Průměrná hodnota počtu přesunů $M_{prům}$ je obtížně určitelná. Lze určit jen její přibližnou hodnotu. $M_{prům} = n(\ln n + \gamma)$ kde $\gamma = 0{,}577216\ldots$ [1]

Lze však říct, že třídění přímým výběrem je efektivnější, než třídění přímým vkládáním. Čím více je zdrojová posloupnost uspořádaná, tím více se tato efektivnost projeví na čase třídění.

3.3 Třídění přímou výměnou

Algoritmus třídění *přímou výměnou* spočívá v postupném porovnávání vždy dvou prostředních prvků a jejich vzájemném prohazování. Pokud třídíme opět od prvku s nejmenším klíčem po prvek s největším klíčem, dojde k prohození vždy, když $a[i]$.klíč >$a[i+1]$.klíč.

Algoritmus třídění přímou výměnou může být buď jednodušší, nebo jej můžeme zlepšit. Jednoduchý algoritmus nazýváme bublinové třídění (Bubblesort) a jeho zlepšením dostaneme algoritmus nazvaný třídění přetřásáním (Shakersort.).

3.3.1 Bublinové třídění (Bubblesort)

Označení *bublinové třídění* může mít dvojí původ. Buď ten, že při porovnávání dvou sousedních čísel vznikne imaginární bublina, v níž se porovnává, nebo takový, že při průchodu algoritmem se postupně nahoru (obrazně) dostávají jednotlivá čísla od těch nejmenších po nejvyšší. Při trošce fantazie může tento proces připomínat bublinky stoupající k hladině.

Bublinové třídění provádíme po krocích, kterých bude n–1. V každém kroku od konce množiny až po prvek i, který představuje číslo kroku porovnáváme nejprve klíč prvku n s klíčem prvku n–1. Poté klíč prvku n–1 s klíčem prvku n–2… Pokud platí, že $a[j]$.klíč < $a[j–1]$.klíč, prvky se prohodí. Je tedy zřejmé, že po jednom kroku bude na pozici i (=číslo kroku) nejmenší číslo celé kontrolované podmnožiny i-n.

Příklad jednoho kroku (porovnávané prvky jsou podtržené):

5, 4, 2, 6, <u>7, 3</u>	porovnáme čísla 7 a 3; protože 7>3, dojde k prohození
5, 4, 2, <u>6, 3</u>, 7	
5, 4, <u>2, 3</u>, 6, 7	porovnáme čísla 2 a 3; protože 2<3, k prohození nedojde
5, <u>4, 2</u>, 3, 6, 7	porovnáme čísla 4 a 2; protože 4>2, dojde k prohození
<u>5, 2</u>, 4, 3, 6, 7	
2, 5, 4, 3, 6, 7	stav posloupnosti po jednom kroku.

Příklad ukázal, že po dokončení kroku 1 je na pozici 1 opravdu nejmenší z porovnávaných čísel. Tento prvek se již nikam přesouvat nebude a jeho pozice je tedy konečná. V kroku 2 se proto budeme zabývat již jen čísly 2-n a v i-tém kroku pouze čísly i-n. V i-tém kroku bude tak počet porovnávání roven n–i.

Počet kroků je n–1 a v každém kroku indexujeme jednotlivá porovnávání písmenkem j nabývajícím odzadu hodnot od n–1 do i. Hodnota j představuje vždy index prvku

z dvojice porovnávaných, a to ten s nižším indexem. Porovnáváme-li 3. a 4. prvek, hodnota *j* bude rovna 3.

Algoritmus bublinového třídění pak vypadá následovně:

```
procedure bublinovetrideni;
    var i, j:integer; x:prvek;          x je pomocná proměnná, kterou využijeme při prohazování prvků, proto musí být stejného druhu jako prvky množiny
begin
  for i:= 1 to n–1 do                   i je číslo kroku a zároveň pozice, na níž bude po dokončení kroku prvek s nejmenším klíčem z prvků i-n
    for j:= n–1 downto i do             j je index prvního ze dvou porovnávaných prvků
      if a[j].klíč > a[j+1].klíč then   porovnávání prvků
                          begin
                            x:= a[j+1];     prohození prvků a[j] a a[j+1]; k tomu
                            a[j+1]:= a[j];  potřebujeme pomocnou proměnnou x
                            a[j]:= x
                          end
end
```

Bublinové třídění naší ilustrativní množiny by po jednotlivých krocích vypadalo následovně:

	i = 1	*i* = 2	*i* = 3	*i* = 4	*i* = 5
5	→2	2	2	2	2
4	5	→3	3	3	3
2→	4	5	→4	4	4
6	→3→	4→	5	5	5
7	6	6	6	6	6
3→	7	7	7	7	7

Z tabulky je možné si všimnout, že počínaje krokem 4 už nedochází k žádnému prohazování. Třídění by tedy mohlo již skončit, ale probíhá dál až po *i* = *n*–1 = 5. Pokud bychom si tedy dokázali zapamatovat, že v určitém kroku se již žádné dva prvky neprohazovaly, mohli bychom program po setřídění zastavit, aby neběžel naprázdno. Pro toto vylepšení použijeme proměnnou typu boolean, která nám bude říkat, zda k prohození v daném kroku došlo či ne. Proměnnou můžeme pojmenovat například *vymena*. Pokud dojde k prohození, proměnné *vymena* se přiřadí hodnota true.

Protože víme, že po vykonání *n*–1 kroků je množina již setříděná, k ukončení cyklu by nám mohla postačit pouze podmínka o setřídění cyklu.

Pozn.: proměnnou *vymena* je třeba před každým krokem vynulovat. Kdybychom tak neučinili, podmínka by platila vždy, pokud by se aspoň jednou v celém třídicím procesu prohazovalo, a třídění by tedy nikdy nezastavila.

Většina množin je utříděná dříve, než v *n*–1 krocích. Pro ideálně neutříděné množiny se však může stát, že se bude ještě i v *n*–1-tém, kroku prohazovat.

Pokud by se v *n*–1-tém kroku ještě prohazovalo, podmínce *vymena* se přiřadí hodnota „true“ a program bude provádět ještě i *n*-tý krok, který však, jak víme, je zbytečný. Můžeme proto třídění zastavit rovněž podmínkou, že *i* <= *n*–1. Kdybychom tak neučinili, program může ve speciálních případech pokračovat i krokem *j* = *n*. Pokud by měl nastat i *n*-tý krok, výsledky by to však nijak neovlivnilo, protože cyklus „for *n*–1 downto *i*“ by co do hodnot vypadal jako „for *n*–1 downto *n*“, a protože *n*–1<*n*, neprovedl by se ani jeden průchod.

Přestože pravděpodobnost toho, že program bude i v *n*-1-tém kroku prohazovat, a podmínka *vymena* nás tedy pustí i do kroku *i*=*n*, sice není vysoká, nelze ji však úplně vyloučit. Ideálním případem rozumíme množinu, v níž by byl prvek s nejvyšším klíčem na prvním místě. Jak jsem již zmínil, vypuštěním podmínky *i* <= *n*–1 nedojde ke změně výsledků a psát ji můžeme jen proto, abychom dosáhli čistoty algoritmu. Znamená však spíše nevýhodu, protože v každém kroku dochází k jednomu porovnávání navíc.

Kompletní program s úpravou ve formě přidání zastavovací proměnné *vymena* pak vypadá následovně:

```
procedure bublinovetrideni2;
  var
  i,j:integer; x:prvek; vymena:boolean;
begin
  i:= 1;
  repeat vymena:= false;
         for j:= n–1 downto i do
         if a[j]>a[j+1]
         then   begin    x:= a[j];
                a[j]:= [j+1];
                a[j+1]:= x;
                vymena:= true;        @ prohazování si zapamatujeme díky proměnné
                                        vymena
                end;
         i:= i+1;                     počítadlo kroků
  until not výmena
end
```

Algoritmus má ještě jednu drobnou nevýhodu: Pokud dojde v jednom kroku ke *k* prohozením, proměnné *vymena* se bude hodnota „true“ přiřazovat *k*-krát, což je zbytečné. Pro naše účely stačí zapamatovat si jediné prohození, což už svědčí o tom, že algoritmus ještě není setříděný. Pokud vezmeme v úvahu, že proces porovnávání zabírá méně času než proces přiřazování, je výhodné řádek označený @ („*vymena* := true;“) nahradit řádkem „**if not** *vymena* **then** *vymena* := true;“. Tím máme zajištěno, že se proměnné *vymena* přiřadí hodnota „true“ pouze jednou.

3.3.2 Třídění přetřásáním (Shakersort)

Algoritmus bubblesort je však výhodný pouze za určitých podmínek.
Např.:

Mějme množinu 6, 1, 2, 3. Čísla v množině jsou téměř uspořádaná. Pokud množinu necháme utřídit programem bubblesort tak, jak jsme ho v předchozí podkapitole navrhli,

program bude v jednotlivých krocích postupovat opět odzadu. Výsledky po jednotlivých krocích:

1. krok 1, 6, 2, 3
2. krok 1, 2, 6, 3
3. krok 1, 2, 3, 6

Z toho je zřejmé, že algoritmus množinu správně uspořádá ve třech krocích, přičemž v každém kroku se provede jediné prohození. Je však patrné, že kdyby byl algoritmus nastaven obráceně, tedy že by v kroku postupoval od čísla s nejnižším indexem po číslo s tím nejvyšším, všechna tři prohození by proběhla v jediném kroku.

Malé číslo z konce množiny se tedy na své místo dostane mnohem rychleji, než velké číslo ze začátku množiny. Protože pravděpodobnost obou možností je stejná, nemůžeme říci, že je bublinové třídění vhodné vždy. Převedením algoritmu na *shakersort* však docílíme toho, že nový algoritmus bude výhodný pro obě možnosti stejně.

Jak program vylepšit, je zřejmé. Dosáhneme toho tím, že budeme neustále prohazovat směry postupu v jednotlivých krocích, tj. jednou odzadu dopředu, podruhé odpředu dozadu.

Další možnost vylepšení programu můžeme vyvodit z následujícího příkladu:

Počáteční množina: 1, 2, 6, 8, 3
1. krok (postup odzadu): 1, 2, 3, 6, 8

V prvním kroku poslední prohození nastalo mezi pozicemi 3 a 4, přesto program pokračoval až do pozice 1. V příštím kroku by hledal nejmenší číslo mezi čísly na pozicích 2-5. Když však víme, že poslední prohození bylo mezi pozicemi 3 a 4, znamená to, že od této pozice je program již setříděn. Stejně tak víme, že číslo na pozici 3 je nejmenší z čísel na pozicích 3-5. Můžeme tedy v dalším kroku hledat nejmenší číslo pouze mezi prvky z pozic (3+1 =) 4 až 5.

V programu to pak znamená, že si musíme pamatovat místo posledního prohození. Budeme vědět, že od toho čísla dále (včetně) je množina již uspořádaná a nemá cenu se jí dál zabývat.

U programu bubblesort jsme použili k jeho zastavení podmínku *vymena* = 0. Nyní se budeme pohybovat ve dvou cyklech v jednom kroku. Např. pro první krok to bude cyklus od n do 1 a od *k*+1 do *n*–1, kde *k* je místo posledního prohození. Jednotlivá místa posledních prohození si budeme pamatovat. Při cestě nahoru začneme od místa *k*+1, kde *k* je místo posledního prohození při předchozí cestě dolů. Při cestě dolů začneme od čísla *k*–1, kde *k* je místo posledního prohození při předchozí cestě nahoru. Lze říci, že se budeme pohybovat vždy v intervalu *l*-*r*. Třídění končí, když bude *l*=*r*, a tedy množina tříděných čísel bude prázdná.

Úplný algoritmus třídění shakersort pak vypadá následovně [1]:

```
procedure shakersort;
    var
        j, k, l: integer; x: prvek;
begin
    l:= 2; r:= n; k:= n;                    vstupní hodnoty pro první krok
    repeat
```

```
    for j:= r dowto l do                    * l-r je aktuálně nesetříděná množina
        if a[j–1].klíč>a[j].klíč then       porovnávání dvou sousedních prvků
        begin
            x:= a[j–1]; a[j–1]:= a[j]; a[j]:= x;   záměna dvou sousedních prvků
            k:= j                           zapamatujeme si číslo výměny
        end;
    l:= k+1;                                v proměnné k je uloženo místo
                                            poslední výměny; v l pak pozice
                                            první nesetříděné proměnné
    for j:= l to r do                       obrácený cyklus cyklu jako *; postup
                                            je od nižších pozic k vyšším
      if a[j–1].klíč>a[j].klíč then begin
                                  x:= a[j–1];
                                  a[j–1]:= a[j];
                                  a[j]:= x;
                                  k:= j
                                end;
    r:= k–1                                 r je pozice poslední nesetříděné
                                            proměnné
  until l>r                                 podmínka ukončení cyklu
end
```

Pokud bychom provedli analýzu třídění přímou výměnou, dostali bychom výsledky, které říkají, že všechna naše vylepšení nepřinášejí až takový efekt, jak bychom očekávali.

V algoritmu bubblesort je počet porovnávání $C = ½(n^2 – n)$ a počty přesunů jsou:
$M_{min} = 0$
$M_{prům} = ¾(n^2 – n)$
$M_{max} = 3/2(n^2 – n)$

Pokud jde o třídění přetřásáním, počty porovnávání jsou tady různé v závislosti na pozicích posledních výměn. *„Nejmenší počet porovnávání je $C_{min} = n–1$. Bylo zjištěno, že průměrný počet přechodů je úměrný $n–k_1\sqrt{n}$ a průměrný počet porovnávání $½[n^2–(k_2+ln\ n)]$.*“[1 - citace ze str. 105]

Všechny naše vylepšení algoritmu bublinového třídění však nijak neovlivňují počet přesunů. Pouze minimalizují počet nadbytečných porovnávání klíčů. Záměna dvou prvků je však, jak víme, mnohem časově náročnější operace, než pouhé porovnávání prvků. Z tohoto hlediska může být naopak algoritmus shakersort ještě náročnější, než bubblesort. Pokud budeme porovnávat porovnávací průchody bubblesortu s a shakersortu, zjistíme, že u třídění přetřásáním dochází kromě případné výměny ještě k přesunům pozice výměny do proměnné *k*. Po skončení cyklu **for** dojde rovněž k přiřazení *l*:= *k*+1 popř. *r*:= *k*–1. Je otázkou, jestli tyto přesuny budou v součtu zabírat více času než nadbytečná porovnávání u bublinového třídění.

Lze však jistě říci, že tolik promyšlené úpravy algoritmu bubblesort nepřinesou takové zlepšení a zefektivnění, jako jsme očekávali. Přesto je ve většině případů výhodnější přece jen použít shakersort namísto bubblesortu.

Z analýz je však patrné, že algoritmy přímé výměny nedosahují ani z daleka takových výsledků, jako algoritmy přímým vkládáním a přímým výběrem. Shakersort je vhodný pro téměř uspořádané množiny, kterých se však v praxi vyskytuje velmi málo.

4. Optimalizace třídění

Přímé metody, kterými jsme se zabývali doposud, posouvají při třídění prvek vždy jen o jedno místo. Např. v množině 6, 1, 2, 3, 4 musí dojít ke 4 přesunům, aby se číslo 6 dostalo na své místo. Kdybychom ale určitým způsobem změnili velikost množiny, v níž se třídí, mohly by to být jen kroky dva, popřípadě dokonce jen jeden.

4.1 Třídění se zmenšováním kroku (Shellovo třídění)

Jedná se o optimalizované třídění přímým vkládáním.

Změnou velikosti množiny, ve které se třídí, míníme její rozdělení na určitý počet podmnožin. Jednu námi zkoumanou množinu vždy rozdělíme na dvě podmnožiny, které můžeme dál dělit stejným způsobem na další podmnožiny. V rámci jedné výsledné podmnožiny prvky setřídíme tříděním přímým vkládáním.

Toto dělení demonstrujme na příkladu:

31	24	95	42	68	18	29	37	třídicí množiny jsou 4 a jsou dvouprvkové: 31 a 68; 24 a 18; 95 a 29; 42 a 37
		třídění s krokem 4↓						
31	18	29	37	68	24	95	42	třídicí množiny jsou 2 a jsou čtyřprvkové: 31, 29, 68, 95 a 18, 37, 24, 42
		třídění s krokem 2↓						
29	18	31	24	68	37	95	42	teprve teď kroky setřídíme přímo, všechny jako jednu množinu
		třídění s krokem 1↓						
18	29	24	31	37	42	68	95	

Nastává další otázka: Jak volit jednotlivé kroky?

Po jednoduchých úvahách zjišťujeme, že je nevhodné volit kroky, které jsou navzájem svými násobky. Kdybychom například postupovali jako v příkladu výše, poznali bychom, že určité prvky jsme mezi sebou porovnávali vícekrát a zbytečně. Pokud jednotlivé kroky nebudou svými násobky, pravděpodobnost porovnávání dvou prvků vícekrát je mnohem menší.

Protože se stále jedná o třídění přímým vkládáním, pořád procházíme množinou a porovnáváme dva prvky. Ne však ty sousední, jako tomu bylo u prostého, ale dva vzdálené od sebe o krok *k*. Pokud klíč prvku *x*, prvku, pro který aktuálně hledáme místo, je menší, než klíč prvku *a*[*j*], přiřadíme prvku *a*[*j*+*k*] hodnotu prvku *a*[*j*] a následně se přesuneme o *k* dále příkazem *j*:= *j*–*k*. Takto postupujeme, dokud bude platit, že *x*.klíč<*a*[*j*]klíč. Pokud podmínka neplatí, znamená to, že na pozici *j* již není větší hodnota a místo pro zařazení prvku *x* má index *j*+*k*.

Nabízí se otázka, co když je v prvku *x* aktuálně nejmenší prvek z prohledávané podmnožiny. Vyhledávání vhodného místa zastavíme tzv. zarážkou. Zarážka je prvek s prvním záporným indexem, jenž můžeme posuzovat v aktuálním krokování:

Např. krok *k*=3. Aktuálně porovnáváme prvky 7, 4, 1. Zarážku stanovíme jako prvek s indexem -2 a hodnota zarážky bude rovna hodnotě prvku *x*. Jestliže by hrozilo, že posloupnost překročí nulu a budou se porovnávat fiktivní prvky se zápornými indexy, podmínka *a*[*j*].klíč<*x*.klíč nebude platit. Ve své podstatě bude vypadat: *x*.klíč<*x*.klíč, což pravda není.

Je tedy zřejmé, že zarážku budeme postupně zvyšovat. Např. třídění s krokem 4:

i	Indexy prvků			Index zarážky	
1	9	5	1	–3	-index první možné zarážky má hodnotu –*k*+1
2	10	6	2	–2	
3	11	7	3	–1	
4	12	8	4	0	
5	13	9	5	1	

Ze řádku 5 vidíme, že zarážka by měla index 1 a pokud bychom prvku *a*[1] přiřadili hodnotu prvku *x*, přepsali bychom si její původní hodnotu. To nesmíme připustit, a proto pokaždé, když by měl index zarážky překročit nulu, vrátíme jej zpět na první možný index tedy na –*k*+1.

Pozn. Je také zřejmé, že při každém takovémto vracení se zvětší počet zkoumaných prvků o jeden. Řádek 5 tedy v programu bude vypadat takto:

13 9 5 1 –3 (zarážka)

Celý algoritmus třídění se zmenšováním kroku pak vypadá následovně:

```
procedure shellsort;
  var i, j , k , s, m, t: integer; x:prvek;
      h: array [1..10] of integer;          pro hodnotu každého kroku potřebujeme
                                            jednu proměnnou; počítáme s tím, že se
                                            nebude třídit více jak deseti různými kroky
begin
  readln (t);                               určení počtu kroků
  if t>1 then readln (h[1]);
  if t>2 then
     for i:= 2 to t–1 do
        while h[i]>h[i–1] do readln(h[i]);  zadávání kroků; ošetřeno tak, aby byly kroky
                                            zadávány od největšího po nejmenší
  h[t]:= 1;                                 poslední krok musí být roven 1

  for m:= 1 to t do
     begin
       k:= h[m];
       s:= –k;                              pozice zarážky v nultém kroku
       for i:= k+1 to n do
          begin
            x:= a[i]; j:= i–k;              počátek třídění přímým vkládáním
            if s=0 then s:= –k;             posun zarážky zpět na zarážku s indexem
                                            jako v nultém kroku
            s:= s+1;                        index zarážky postupně zvyšujeme
            a[s]:= x;                       zarážka musí mít hodnotu jako zkoumaný
                                            nezařazený prvek
            while x.klíč<a[j].klíč do
                  begin
                    a[j+k]:= a[j];          třídění přímým vkládáním s krokem k pro
                                            prvky a[s+z*a], kde z je libovolné přirozené
                                            číslo takové, aby platilo s+z*a<=n
```

```
                j:= j–k
              end
          a[j+k]:= x
        end
    end
end
```

Řekli jsme, že kroky by neměly být navzájem násobky. Při deklaraci jednotlivých kroků bychom mohli tedy zavést kromě podmínky $h[i]>h[i–1]$ také podmínku $h[i–1]/h[i] = h[i–1]$ div $h[i]$, logickým operátorem by byl „**or**“. Význam je následující: dokud je $h[i]$ větší než $h[i–1]$ nebo dokud bude $h[i–1]$ násobkem $h[i]$, tedy hodnoty prostého a celočíselného dělení jsou si rovny.

Analýza třídění vkládáním se zmenšováním kroku je velice obtížně matematicky popsatelná. Závisí na počtu kroků, hodnotách kroků, přičemž pro různé množiny se tyto aspekty mohou značně lišit, a to až natolik, že výhodnější než Shellsort bude obyčejné třídění s přímým vkládáním (a jediným krokem 1).

Platí, že pokud utřídíme posloupnost s krokem k, pak tato zůstává po celý zbytek třídění utříděna s krokem k. Platí pouze, pokud krok m > krok m+1.

Doporučují se (Knuth) tyto kroky(psáno od konce):
1, 4, 13, 40, 121,… kde $h[m–1]=3h[m]+1$, počet kroků $t = \log_3 n – 1$
nebo
1, 3, 7, 15, 31,… kde $h[m–1]=2h[m]+1$, počet kroků $t = \log_2 n – 1$

„Matematickou analýzou druhé alternativy Shellova algoritmu třídění n-prvkové posloupnosti se zjistilo, že jeho složitost je úměrná $n^{1,2}$,“ [1 - citace ze str. 108] což je lepší hodnota než n^2.

4.2 Třídění rozdělováním

Třídění rozdělováním, jinak též *quicksort*, vychází z třídění přímou výměnou. Jedná se o jeden z nejúčinnějších třídicích algoritmů, a proto je s podivem, že vychází z algoritmu, který je tak málo efektivní. Výměny zde však probíhají na velké vzdálenosti.

Ústřední část programu spočívá opět v určitém porovnávání a výměnách. V quicksortu tyto děje probíhají následovně:

Mějme určitou nesetříděnou posloupnost. Např.:

4	8	2	5	11	7	3	9

Nyní vybereme určitý prvek. Můžeme tak učinit náhodně, ale výhodnější je vybrat prvek s indexem n div 2, tedy prvek 5. Tímto prvkem rozdělíme množinu na dvě části. Na tu, která bude obsahovat prvky menší než 5 a větší než 5.

Nyní budeme procházet cyklus od začátku (krokujeme proměnnou i) i od konce (krokujeme proměnnou j) a v obou případech se zastavíme tehdy, pokud nalezneme nesrovnalost v pořadí prvku $a[i]$ a prvku 5: Víme, že když se pohybujeme od začátku, tedy v intervalu před prvkem 5, měli bychom míjet pouze prvky menší než 5. Nesrovnalost znamená, že narazíme na prvek větší než 5. U tohoto prvku se zastavíme a pokračujeme

v hledání další nesrovnalosti od konce. Zde naopak víme, že bychom měli míjet pouze prvky vyšší než 5. Nesrovnalost znamená, že narazíme na prvek menší než 5.

Ilustrujme si toto vyhledávání na našem případu: Nejprve postupujeme množinou od začátku, tedy uvažujeme prvky, které by měly být menší než 5. Pro prvek *a*[1]=4 toto platí. Ale u prvku *a*[2] nalézáme nesrovnalost. Jeho hodnota je 8 a tedy prvek patří do druhé poloviny množiny. Tam jej také zařadíme a sice na pozici první nesrovnalosti, kterou v této druhé polovině objevíme. Zde by naopak měly být prvky větší než 5. Pro prvek *a*[8]=9 to platí. V prvku *a*[7]=3 však nalézáme nesrovnalost. Víme, že prvky s indexy 2 a 7 jsou v opačných polovinách množiny. Není nic jednoduššího, než je navzájem prohodit, aby se tak každý dostal do správné poloviny.

Dostaneme tedy množinu:

4 3 2 5 11 7 8 9

Dále můžeme pokračovat ve vyhledávání od následujících prvků, tedy od prvku *a*[3]=2 a *a*[6]=7, protože víme, že pro tyto prvky již nesrovnalosti nenastanou.

Nabízí se otázka, co se stane v případě, že se oba cykly, tedy ten od začátku a ten od konce navzájem minou. V takovém případě by došlo zcela jistě k nežádoucím výměnám. Aby cyklus přestal vyhledávat nesrovnalosti, ošetříme ho proto zastavující podmínkou *i*>*j*. Protože v algoritmu použijeme cyklus **repeat**, může i přesto dojít k jednomu chybnému prohození. Prohození tedy povolíme pouze v případě, že *i*<=*j*. Pokud *i*=*j*, jedná se o jeden a týž prvek. Prohození tedy nemá význam. Avšak význam mají následné dva příkazy, které zvýší *i* (případně sníží *j*) o 1, tedy cyklus **repeat** se zastaví díky již zmíněné podmínce *i*>*j*.

Tento podprogram algoritmu quicksort vypadá následovně [1]:

```
procedure rozdel;
    var w, x: prvek;
begin
  i:= 1; j:= n;
  x:= a[(i+j) div 2];                  prvek můžeme zvolit i náhodně
  repeat
    while a[i].klíč < x.klíč do i:= i+1;   hledání nesrovnalosti od začátku
    while x.klíč < a[j].klíč do i:= j–1;   hledání nesrovnalosti od konce
    if i<=j then                         podmínka brání tomu, aby se prohodily prvky
                                         poté, co se cykly od začátku a od konce minou
      begin
        w:= a[i]; a[i]:= a[j]; a[j]:= w;
        i:= i+1; j:= j–1;                indexy posuneme dále, prvky i a j už netřeba
                                         prověřovat
      end
  until i>j
end
```

Máme tedy celou naši množinu rozdělenou na dvě poloviny. První je větší než hodnota proměnné *x*, druhá je menší. Tím se nám zároveň množina z části setřídila. Obdobný třídicí proces můžeme použít na obě nově vzniklé poloviny, pokud tedy ověříme, že je to ještě třeba.

Při každém tomto dílčím procesu vzniknou dvě nové podmnožiny, v rámci kterých pak budeme opět obdobně třídit. Takto postupně půlíme celkovou množinu na podmnožiny, a to nejvýše tak dlouho, dokud nevzniknou jednoprvkové množiny, v nichž už k žádnému prohazování nedojde. Tím skončí třídění jedné takovéto větve.

Uveďme si příklad:

5 7 2 6 8 3 1 4

vybereme prvek s indexem 1+8 div 2 = 4 a tím množinu rozdělíme a částečně setřídíme:

5 4 2 1 3 8 6 7

rozdělili jsme tedy množinu na dvě části, v rámci kterých dále třídíme:

5 4 2 1 3 a 8 6 7

opět zvolíme prostřední prvek a třídíme:

1 2 4 5 3 6 8 7

opět se nám podmnožiny rozdělily na další podmnožiny, v rámci kterých třídíme.

Vlastní algoritmus třídění bude pokračovat sám sebou, avšak se vstupními hodnotami dané podmnožiny, kterou budeme třídit. Teprve až zjistíme, že už není třeba podmnožinu třídit, větvení přestane a algoritmus přejde k třídění nejbližší sousední větve.

Algoritmus tedy musí být psaný jako procedura, jinak bychom jej nemohli v rámci sebe sama vyvolat. Samotný program bude obsahovat pouze jediné vyvolání této procedury se vstupními hodnotami celé naší velké nesetříděné množiny. Jevu, kdy procedura obsahuje sama sebe, říkáme *rekurze*.

Kompletní algoritmus třídění quicksort vypadá takto:

```
procedure quicksort;
  procedure trid (l, r: integer);
    var i, j: integer; w, x: prvek;
  begin
    i:= l; j:= r;
    x:= a[(l+r) div 2];
    repeat
        while a[i].klíč < x.klíč do i:= i+1;
        while x.klíč < a[j].klíč do i:= j–1;
        if i<=j then
            begin
              w:= a[i]; a[i]:= a[j]; a[j]:= w;
              i:= i+1; j:= j–1;
            end
    until i>j;
    if l<j then trid (l, j);          pokud nedošlo j až po nejnižší možný index
                                      l, je třeba utřídit tuto dolní polovinu tříděné
                                      množiny
    if i<r then trid (i, r);          obdobně to platí i pro druhou polovinu
  end
begin trid (1, n)                     vlastní program; pouze vyvoláme proceduru
end
```

„Abychom mohli uskutečnit analýzu algoritmu quicksort, musíme nejprve přezkoumat proces rozdělení pole na úseky. Volbou prvku x dojde k přeuspořádání celého pole, takže

potřebujeme přesně n porovnání. Počet výměn prvků pole určíme na základě pravděpodobnosti následující úvahou:

Předpokládejme, že prvky pole, které máme rozdělit na dva úseky, obsahují n klíčů: 1-n. Vybereme nějaký prvek x. Po rozdělení pole na úseky se bude prvek x nacházet v poli na x-té pozici. Počet potřebných výměn se bude rovnat součinu počtu prvků v levém úseku (x–1) a pravděpodobnosti výskytu klíče, který je třeba vyměnit. Klíč se vymění, když není menší než prvek x. Tato pravděpodobnost je (n–x+1)/n. Očekávaný počet výměn pro všechny možné volby prvku x je potom daný vztahem:

$$M = \frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}\cdot(n-x+1) = \frac{n}{6} - \frac{1}{6n}$$

Vidíme, že očekávaný počet výměn je přibližně roven n/6.“ [1- citace ze str. 121]

Pokud při každé volbě *x* vybere program vždy medián, tj. prostřední hodnotu množiny, rozpadne se množina na dvě poloviny. Počet přechodů se v tomto případě bude rovnat log *n*, celkový počet porovnání bude *n* . log *n* a počet potřebných výměn *n*/6 . log *n*.

Přirozeně se nám podaří vybrat medián jen výjimečně s pravděpodobností 1/*n*. Průměrná účinnost je však v porovnání s tou ideální horší pouze o faktor 2.ln 2.

Pokud jde o nejhorší možný případ, ten nastane, pokud se dělením pole o *n* prvcích vždy rozdělí na dvě podmnožiny, první o *n*–1 prvcích a druhá o jediném prvku. Místo log *n* rozdělení potřebujeme tedy *n* rozdělení a účinnost je v tomto případě rovna n^2. [1]

5. Závěr

Seznámili jsme se s několika základními a optimalizovanými třídicími algoritmy. Zmínil jsem jejich výhody i nevýhody odvíjející se od posloupnosti prvků nesetříděné množiny. Pokud budeme mít více informací o množině, kterou je třeba setřídit, můžeme s ohledem na to volit i příslušný třídicí algoritmus.

Základní algoritmy jsou jednoduché na pochopení, avšak nepůsobí tak efektivně, jako optimalizované, které se z nich odvíjejí.

Nutno však poznamenat, že i přes veškerou naši snahu nemají volby třídicích algoritmů hlubší význam. Potřebujeme-li setřídit například množinu o tisíci prvcích, pro dnešní počítače je to otázka několika sekund, bez ohledu na to, jaký třídicí algoritmus k tomu použijeme. Ve zkratce lze říci, že ztráta času, kterou bychom vynaložili na úvahu o tom, jaký algoritmus zvolit, je mnohem vyšší, než ztráta času způsobená případným nevhodným zvolením algoritmu.

Jestliže však na principu algoritmů třídění třídíme něco my manuálně, nebo stroj s reálnými a nebo virtuálními objekty, má už řeč o tomto tématu hlubší smysl. V takových případech je důležité si správně uvědomit, o jaké prvky se jedná a jak jsou uspořádány a podle toho zvolit příslušný algoritmus. Je totiž možné, že zdánlivě málo dokonalý algoritmus utřídí množinu mnohem rychleji, než jiné vylepšené algoritmy, a to právě díky specifickému uspořádání prvků množiny.

Rovněž pokud je třídicí algoritmus součástí nějakého vyššího programu, který bude obsahovat mnoho příkazů k utřídění množiny, je třeba se otázkou těchto algoritmů více zabývat.

Pomocí symbolů *C* (počet potřebných porovnání), *M* (počet potřebných přesunů) a *n* (počet porovnávaných prvků) můžeme analyzovat všechny tři přímé třídicí metody (*tab. 1*). *„Jednotlivé sloupce této tabulky obsahují údaje o minimálních (min), průměrných (prům) a maximálních (max) počtech porovnání a přesunů prvků uvažovaných pro všechny n! permutací prvků.“* [1 - citace ze str. 127]

Tab. 1. Porovnání přímých metod třídění[1]

		min	*prům*	*max*
Přímé vkládání	*C*	$n - 1$	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	M	$2(n - 1)$	$(n^2 - 9n - 10)/4$	$(n^2 + 3n - 4)/2$
Přímý výběr	*C*	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	M	$3(n - 1)$	$n(\ln n + 0{,}57)$	$n^2/4 + 3(n - 1)$
Přímá výměna - bubblesort	*C*	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	M	0	$(n^2 - n) \cdot 0{,}75$	$(n^2 - n). 1{,}5$

„Pro přepracované metody vnitřního třídění nejsou známé takové jednoduché a přesné vzorce. Z analýz těchto algoritmů vyplývá, že výpočtová složitost Shellova třídění je $c_1 \cdot n^{1,2}$ *a* $c_1 \cdot n \cdot \log(n)$ *v případě třídění quicksortem.*

Uvedené vzorce poskytují přibližnou míru účinnosti algoritmů třídění. Udávají efektivnost jako funkci celkového počtu prvků n a umožňují dělení třídících algoritmů na jednoduché přímé metody se složitostí n^2 *a přepracované logaritmické metody se složitostí* $n \cdot \log n$.“ [1 - citace ze str. 127]

6. Použitá literatura

[1] WIRTH, Niklaus. *Algoritmy a štruktúry údajov*. Pavol Fischer. Bratislava : Západoslovenské tlačiarne, 1988. 488 s.

[2] PADRTA, David. *Algotitmy a programování* [online]. 2006 [cit. 2008-05-20]. Dostupný z WWW: <http://sweb.cz/david.padrta/pascal/alg_prog.html>.

[3] *Pascal pro experty* [online]. 2003 [cit. 2008-05-20]. Dostupný z WWW: <http://programar.webpark.cz/pascal/pexpert.html>

Seznam příloh:

1. CD s vypracovanou bakalářskou prací ve formátu pdf.