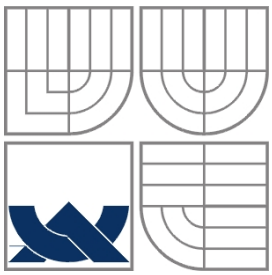


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

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 **Jiří Lýsek**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. Jiří Dvořák, CSc.**
SUPERVISOR

BRNO 2008

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

LICENČNÍ SMLOUVA

ABSTRAKT

Práce se zabývá problémem třídění polí, který patří mezi známé problémy informatiky. Definuje pojmy třídění, algoritmus a složitost algoritmů. Popisuje vybrané algoritmy pro třídění od nejjednodušších po vybrané složitější konstrukce. Nakonec jsou algoritmy porovnány v závislosti na počtu operací a době trvání celého procesu.

ABSTRACT

This thesis deals with a problem of sorting arrays, which is one of known problems in computer science. It defines concepts of sorting, algorithm and algorithm complexity. It describes sorting algorithms from the easiest ones to the selected more difficult methods. Finally, the algorithms are compared according to number of needed operations and time needed for the whole sorting process.

KLÍČOVÁ SLOVA

algoritmus, třídění, složitost algoritmů, bublinové třídění, třídění vkládáním, třídění výběrem, Shellovo třídění, třídění haldou, rychlé třídění

KEYWORDS

algorithm, sorting, algorithm complexity, bubblesort, insertionsort, selectionsort, shellsort, heapsort, quicksort

Obsah:

1 ÚVOD

Třídění jako takové je pro nás téměř univerzálně vykonávanou činností. Se setříděným souborem dat se setkal naprosto každý, ať se jednalo o telefonní seznam nebo obsah knihy nebo slovník cizích slov. Tyto operace byly většinou vykonávány v počítačích, nicméně v dnešním počítačovém věku, kdy lidé v rozvinutém světě používají počítač téměř denně, je to také setříděný seznam souborů podle jména nebo data vytvoření, setříděné výsledky vyhledávání z vyhledávače podle relevance či seznam produktů v internetovém obchodě, seřazený například podle ceny.

Tříděním se obvykle rozumí proces přeuspořádání objektů v požadovaném pořadí, podle určitého daného kritéria. Jeho účelem je obvykle ulehčit následné prohledávání dané množiny objektů. Proto je třídění důležité, jak pro člověka, tak pro počítač, aby mohl se seřazenými prvky efektivně a rychle pracovat.

2 ROZBOR PROBLÉMU

Pokud jde o algoritmy třídění, je to typická ukázka, jak se k jednomu výsledku dostat více způsoby. V některých případech je právě volba vhodného algoritmu velmi důležitá. Jde tedy o to, že máme zadanou posloupnost

$S = (S_1, S_2, S_3, \ldots, S_n)$

a hledáme posloupnost

$S = (S'_1, S'_2, S'_3, \ldots, S'_n)$,

pro kterou platí:

1. je seřazená

   $S_1 \leq S_2 \leq S_3 \ \ldots \leq S_n$

2. je permutací zadané posloupnosti S (obsahuje stejné prvky, ale v jiném pořadí)

2.1 Algoritmus

Nejprve osvětlíme pojem algoritmus. Je to vlastně přesný předpis, kterým jde vyřešit daný typ úlohy. Algoritmus může být například i recept k přípravě jídla. V našem případě je to teoretický princip řešení problému, není to tedy ještě přesný zápis v programovacím jazyce. Nezabývá se ani technickými podrobnostmi implementace (například použití ukazatele proti zkopírování obsahu proměnné). Nicméně je to přesný návod jak s daným vstupem pracovat (např. porovnat s jiným nebo někam uložit). Pro člověka by se některé algoritmy mohly zdát příliš podrobné, ale pro sestavení počítačového programu je to nutnost.

2.1.1 Vlastnosti algoritmů

Od algoritmu vyžadujeme určité vlastnosti, podle kterých můžeme hodnotit kvalitu algoritmu [3], [13]:

Konečnost:

Každý algoritmus musí skončit v konečném počtu kroků. Počet takovýchto kroků může být libovolně velký, záleží na rozsahu a velikosti vstupních hodnot, ale pro každý vstup musí být konečný. Existují však i postupy, které tuto podmínku nesplňují, a ty se nazývají výpočetními metodami.

Správnost:

Výsledek algoritmu musí být správný ve vztahu ke vstupním datům a k zadání problému. Nikdo ovšem neručí (pokud to není nějak algoritmem ošetřeno) za to, jestli jsme použili správný algoritmus na získaná data. Například některé výpočtové teorie mají velmi silné zjednodušující předpoklady a když je nedopatřením zanedbáme, dostáváme naprosto scestné výsledky.

Determinovanost:

Každý krok algoritmu musí být jednoznačně a přesně definován. V každém okamžiku běhu algoritmu musí být přesně dáno, co se má udělat, také musí být jasné, jak má provádění algoritmu pokračovat. Běžný jazyk obvykle neposkytuje možnosti jak naprosto přesně a jednoznačně zadat co se má provádět, proto byly pro přesný zápis algoritmů navrženy programovací jazyky. V nich má každý příkaz přesně daný význam. Vyjádření algoritmu v programovacím jazyce se nazývá program. Algoritmus nemusí být zapsán přímo v jazyce interpretovatelném do strojového kódu, ale může být zapsán i pomocí

„pseudokódu“ [15] (podobný jako programovací jazyk, ale zanedbává některé detaily implementace jako deklarace proměnných apod.).

Vstup:
Algoritmus obvykle zpracovává nějaké vstupní hodnoty, aby mohl dojít k výsledku. Tyto vstupy mu mohou být předány před začátkem práce, nebo i v jejím průběhu. Vstupy mají obvykle definovaný typ hodnot jakých mohou nabývat (čísla, řetězce, …).

Výstup, resultativnost:
Algoritmus má alespoň jeden výstup. Je to veličina, která je v nějaké požadované relaci ke vstupům, a tím tvoří odpověď na problém, který algoritmus řeší. Algoritmus vede od zpracování hodnot k výstupu – resultativnost. Výstupní hodnota algoritmu může být i chybové hlášení.

Obecnost (hromadnost):
Algoritmus neřeší jeden konkrétní problém (např. „jak spočítat 5×12“), ale obecnou třídu obdobných problémů (např. „jak spočítat součin dvou celých čísel“).

Při návrhu aplikací jsou proto brány v úvahu hlavně algoritmy, které jsou v nějakém smyslu kvalitní. Algoritmus by měl tedy splňovat určitá kritéria, týká se to například počtu kroků potřebných pro vykonání algoritmu, nebo to může být jednoduchost či elegance algoritmu. S tím také souvisí požadavek na to, aby algoritmus nespotřeboval příliš mnoho operační paměti stroje.

2.1.2 Druhy algoritmů

Klasifikace algoritmů se dá provést mnoha způsoby, a někdy je jeden algoritmus členem i více než jedné takové skupiny [3]:

Rekurzivní algoritmy:
používají samy sebe, dokud nedojdou do určitého konečného stavu. Jako příklad rekurzivního algoritmu je nejčastěji uváděn postup výpočtu faktoriálu. [6]

Hladové algoritmy:
k řešení se propracovávají po jednotlivých rozhodnutích, která, jakmile jsou jednou učiněna, už dále nejsou brána v úvahu.

Algoritmy typu rozděl a panuj:
dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), nakonec se dílčí výsledky vhodným způsobem sloučí.

Algoritmy dynamického programování:
pracují tak, že postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů. Mnoho úloh se řeší převedením na grafovou úlohu a aplikací příslušného grafového algoritmu.

Pravděpodobnostní algoritmy:
provádějí některá rozhodnutí náhodně či pseudonáhodně.

Paralelní algoritmy:
používají se v případě, kdy máme k dispozici počítač s více výpočetními jednotkami, nebo více propojených počítačů (výpočetní cluster) pro řešení daného problému.

Heuristické algoritmy:
nehledají přesné řešení, ale snaží se nalézt vhodnou aproximaci. Používají se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

2.2 Složitost algoritmů

Algoritmy jako takové je potřeba od sebe nějak odlišit. Například u našeho problému třídění existuje velké množství algoritmů, jejichž výstup je sice v podstatě stejný (rozdílem stabilních a nestabilních algoritmů se budeme zabývat později) tj. seřazená množina dat, ale doba jejich trvání, nebo náročnost na systémové prostředky se může několikanásobně lišit.

Z tohoto důvodu se zavádí pojem složitosti algoritmů. Jak bylo naznačeno, rozlišujeme tedy *časovou složitost*, tedy potřebné množství času a *prostorovou složitost*, tedy potřebné množství operační paměti pro datové struktury použité algoritmem. Mezi těmito dvěma druhy složitosti existuje závislost. Počítač může během určitého počtu kroků obsadit pouze určitý počet místa v paměti. Nejde ale určit z časové složitosti prostorovou a obráceně.

Složitost můžeme vyšetřovat:
- exaktně – pomocí matematického aparátu
- neformálně – analýzou programu

2.2.1 Časová složitost

Je obvykle definována jako počet kroků, které je nutné vykonat k dosažení výsledku pomocí daného algoritmu [2]. Obvykle bývá uvedena jako funkce počtu zpracovaných vstupních prvků:

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

Toto označení se nazývá jako „velké O notace“ a označuje horní asymptotu složitosti. Ke klasifikaci složitosti algoritmu se používá nejčastěji.

Algoritmus je potom charakterizován pomocí tří takovýchto funkcí, kde jednotlivé vyjadřují typ složitosti [12]:

Maximální: O
je to horní mez složitosti. Je to složitost pro nejhorší možný tvar vstupních dat pro algoritmus. V našem případě například řazení opačně seřazené posloupnosti. Tato funkce je vždy větší nebo rovna skutečné složitosti při praktickém provádění algoritmu.

Průměrná: Θ
je to složitost, která je přibližně očekávána pro běžný (náhodný) tvar vstupních dat do algoritmu. Je zjištěna na základě pravděpodobnosti výskytu určitého tvaru

vstupních dat.

Minimální: Ω
tato funkce je dolní mez složitosti. Říká nám jakých výsledků algoritmus dosáhne při nejlepším tvaru vstupních dat. Pokud jde o třídění, vstupem může být například již seřazená posloupnost. Hodnota skutečné složitosti nikdy nebude menší než tato minimální složitost.

Ještě lze narazit na symboly „o“ – odhad složitosti ostře shora (o řád více) a „ω“ – odhad složitosti ostře zespodu (o řád méně). Ale nejsou tolik využívané.

2.2.2 Řád složitosti

Algoritmus je klasifikován řádem složitosti, který je charakterizován funkcí $g(n)$ z výrazu $f(n) = O(g(n))$. Řád je určen dominantní složkou funkce $g(n)$, která nejvíce ovlivňuje výsledek.

Př. 2.1.: složitost je dána funkcí:
$g(n)= n^3 + 3n - 3$
říkáme potom že složitost je řádu n^3 a zapíšeme to jako $O(n^3)$

Příklady některých složitostí (od nejlepší k nejhorší):

$O(1)$	konstantní
$O(\log_2 n)$	logaritmická
$O(n)$	lineární
$O(n^c)$, $c > 1$	polynomiální
$O(c^n)$	exponenciální
$O(n!)$	složitost daná faktoriálem

Za rozumně aplikovatelné se obvykle považují maximálně algoritmy s polynomiální složitostí. Další (např. typu $O(n!)$) mohou i při malém navýšení počtu vstupních dat zabrat strojový čas, který se bude lišit od původního i o řád více (z hodin se stanou dny apod.), což je nepřijatelné.

2.2.3 Prostorová složitost

Tento druh složitosti určuje, kolik operační paměti je potřeba pro provedení algoritmu. Udává se obvykle také jako funkce počtu zpracovaných prvků. Nejlepší jsou algoritmy, které jsou schopny pracovat přímo na zpracovávaných vstupních datech a nevyžadují přídavný prostor, nebo případně jen velmi malý. O takovém algoritmu říkáme, že pracuje *„na místě“ („in situ“* nebo anglicky *„in place“)*. Někdy ovšem stojí za zvážení použití algoritmu, který vyžaduje přídavný prostor pro jeho větší efektivitu.

2.3 Třídění

Tříděním se rozumí, jak bylo řečeno v úvodu, seřazení dat podle určitého kritéria. Pro člověka přirozené je například třídění podle abecedy nebo podle nějakého číselného kritéria vztaženého k danému objektu tříděné množiny. Pro počítače je vhodné udržovat setříděná data z toho důvodu, že v nich může snadněji vyhledávat podle požadavků uživatele.

2.3.1 Pole

K pochopení problematiky je ještě třeba objasnit, co vlastně budeme třídit. V počítači jsou obvykle data pro třídění prezentována jako pole [7]:

- Je to datová struktura podobná vektoru (jednorozměrné pole) nebo matici (vícerozměrné pole).
- Pole sdružuje určitý počet prvků stejného typu a má definovanou velikost (tato omezení některé programovací jazyky nekladou, ale budeme předpokládat právě takováto pole).
- Většina programovacích jazyků má pole jako vestavěný datový typ.
- K prvku v poli se lze dostat pomocí indexu – to je většinou celočíselná hodnota, ale některé jazyky dovolují i indexy textové – taková pole jsou pak nazývána jako *pole asociativní*.
- Většina jazyků používá k identifikaci prvku pole hranaté závorky.

Výhodou setříděného pole je, že v něm jde efektivně vyhledávat například metodou půlení intervalu se složitostí $O(\log_2 n)$.

Nyní nadefinujeme prvek a pole *n* prvků, které bude využito v následujících příkladech:

```
TPrvek = record
  klic: integer;
  { další položky prvku pole – vázané hodnoty }
end;

TSeznam = array [0 .. n] of TPrvek;
```

2.3.2 Klíče a hodnoty

Z hlediska řazení je soubor dat chápán jako skupina dvojic klíč ⇔ hodnota. Pro seřazení se používá klíčů a ve výsledku máme monotónní posloupnost klíčů zatímco na připojené hodnoty se při řazení nebere ohled a pouze se přesouvají podle potřeby s klíčem.

V případě pole, které má prvek definován pouze jeho hodnotou, jsou řazeny pouze klíče, které jsou zároveň hodnotami. Na druhé straně stojí pole, které má více klíčů, podle kterých jej lze setřídit. Pro řazení takovéhoto pole je potřeba speciální porovnávací funkce, která vyhodnotí dva prvky podle potřeby s ohledem na všechny klíče podle jejich priority. Druhým způsobem je postupné řazení podle jednotlivých klíčů pomocí více průchodů polem při použití zvoleného algoritmu. V každém dalším průchodu polem je tříděno podle klíče s větší prioritou. Pro takovéto třídění je zapotřebí stabilní algoritmus.

2.3.3 Stabilita algoritmů

Rozdíl mezi stabilním a nestabilním algoritmem zpozorujeme tehdy, když máme v poli více hodnot se stejnými klíči. Jde o to, že pokud ve zdrojovém souboru je prvek A před prvkem B (se stejným klíčem) a po seřazení toto pořadí stále platí, potom je algoritmus stabilní.

Příkladem může být seznam měst, setříděný podle jména. Pokud ho necháme seřadit, podle okresů stabilním algoritmem, zůstane pořadí měst v okresech stejné jako v předchozím způsobu seřazení. Pokud bychom použili nestabilní algoritmus, byla by jména měst v okresech neuspořádaná.

2.3.4 Vzestupné a sestupné třídění

Data lze třídit buď tak, že setříděná posloupnost bude mít první prvek nejmenší a

poslední prvek největší – vzestupně, nebo opačně – tedy sestupně. Obvykle stačí v algoritmu zaměnit smysl porovnávání dvojic dat.

2.3.5 Přirozenost

Přirozený algoritmus je charakteristický tím, že částečně seřazený seznam zpracuje rychleji než neseřazený. A opačně, nepřirozenému algoritmu, lépe vyhovuje posloupnost, která je seřazená proti směru výsledku.

2.3.6 Vnitřní třídění

Je to takový druh třídění, při kterém jsou všechny operace (nezávisí na algoritmu) prováděny v operační paměti počítače. Takovéto uložení dat umožňuje algoritmu okamžitý přístup ke všem datům naráz.

2.3.7 Vnější třídění

Při takovémto třídění jsou tříděná data uchovávána na vnějších záznamových médiích (disk nebo dříve páska) a přístup k nim je omezen rychlostí tohoto média. Další předpoklad je, že soubory jsou na tomto médiu uloženy sekvenčně a my máme přístup pouze k jednomu, nebo omezenému počtu záznamů. Takovýto druh třídění je nevyhnutelný, pokud je objem dat pro setřídění větší než operační paměť stroje. Potom je nutné mít v paměti jen část těchto dat a zbytek uložený na záznamovém médiu.

Problémem je zde také, že většinou dopředu nemáme znalost o velikosti tříděného souboru. Faktorem, na který bychom měli v této situaci brát ohled je také to, že přístup do souboru trvá o mnoho déle než porovnání dvou prvků – na druhou stranu nám zůstává volná většina operační paměti, takže s ní není třeba příliš šetřit.

3 VNITŘNÍ ALGORITMY TŘÍDĚNÍ

3.1 Klasifikace

Tyto algoritmy se používají pro třídění polí přímo v operační paměti počítače. Po těchto algoritmech vyžadujeme, aby byly šetrné na objem využité počítačové paměti (když už v ní mají nahrané celé tříděné pole). Jde tedy o to, aby permutace, kterou prvky vytvářejí, vznikala *na místě* a proto metody, které vytvářejí kopii pole, pro nás nejsou příliš zajímavé.

Dobrou představu o kvalitě algoritmu dostaneme, pokud budeme měřit počet potřebných porovnání klíčů *C* a počet potřebných přesunů prvků *M*. Mezi algoritmy můžeme vypozorovat čtyři základní metody, kterými algoritmus pracuje [1], [2]:

- Třídění výměnou:
  V souboru se vždy najde (nějakou metodou závislou na konkrétním algoritmu) nějaká dvojice prvků, která je ve špatném pořadí, a tyto prvky se navzájem zamění.
- Třídění vkládáním:
  Ze souboru neseřazených dat se postupně bere položka po položce a vkládá se na správné místo v seřazeném souboru.
- Třídění výběrem:
  V souboru se vždy najde nejmenší ze zbývajících nesetříděných položek a uloží na konec postupně budovaného seřazeného souboru (v kterém je tato hodnota největší).
- Třídění rozdělováním:
  Vstupní soubor se rozdělí na části, které se (typicky rekurzivně) seřadí; výsledné seřazené části se poté sloučí takovým způsobem, aby i výsledek byl seřazený.

3.2 Vybrané algoritmy třídění a jejich popis

Metody přímého třídění do sebe zahrnují ty nejjednodušší třídící algoritmy. Jsou to algoritmy (Bubblesort, Insertionsort, Selectionsort), které pracují na místě, jsou krátké a jednoduché. Jejich časová složitost je většinou v řádu $O(n^2)$. Je tedy jasné, že nejsou příliš použitelné pro velké objemy vstupních dat, ale pokud není množství dat příliš velké, není třeba vybírat příliš složitý algoritmus – obzvláště když přihlédneme k rychlosti dnešních počítačů.

Naproti tomu nepřímé, složitější metody jsou schopné běžet i v řádu $O(n \log_2 n)$ a tím jsou pro nás zajímavé (Shellsort, Heapsort, Quicksort).

3.2.1 Bubblesort – třídění přímou výměnou

Tento algoritmus je typickým zástupcem algoritmů používajících metodu třídění výměnou. Je to v podstatě nejjednodušší a nejintuitivnější algoritmus pro třídění vůbec. Jeho princip je takový, že prochází pole stále dokola od začátku do konce. Pro každý prvek provede porovnání s jeho sousedem a pokud nejsou prvky v požadovaném pořadí, zamění je. Pokud v průchodu celým polem neprovede ani jednu záměnu, je jasné, že pole je setříděné a algoritmus končí [8].

Algoritmus se vyskytuje v mnoha variacích a vylepšeních. Přesto je to algoritmus pomalý a neefektivní. Využívá se hlavně pro výukové účely, v jednoduchých aplikacích nebo jednorázově. Jeho implementace je velmi jednoduchá.

Algoritmus pracuje na místě (nevyžaduje pomocnou paměť), je stabilní (prvkům se

stejným klíčem nemění vzájemnou polohu), patří mezi přirozené řadicí algoritmy (částečně seřazený seznam zpracuje rychleji než neseřazený).

Program 3.1. Bubblesort

```
procedure BubbleSort;
var
  zameneno: boolean;
  i: integer;
begin
  repeat
    zameneno := false;
    for i := 0 to length(seznam) - 2 do begin
      if seznam[i].klic > seznam[i + 1].klic then begin
        Zamena(seznam[i], seznam[i + 1]);
        zameneno := true;
      end;
    end;
  until not zameneno;
end;
```

Analýza třídění výměnou:

Algoritmus se dá ještě vylepšit například tím, že si budeme pamatovat pozici poslední výměny a v každém dalším průchodu hlavním cyklem za tuto pozici již nebudeme zacházet, protože za ní jsou data již setříděná. Značně se tím sníží počet kroků s porovnáváním.

Zajímavé na tomto algoritmu je, že je velice rychlý pro již setříděné posloupnosti. Je to dáno tím, že pro takovou posloupnost algoritmus nemusí ani jednou prohazovat pořadí položek a po prvním průchodu posloupností končí. Další zvláštnost je, že pokud je největší položka na začátku dat, dostane se na své místo na konci tříděné posloupnosti během prvního průchodu algoritmu přes posloupnost, naopak položka s nízkou hodnotou klíče, na konci posloupnosti, bude procházet na své místo jen velmi pomalu.

Z toho plyne další možnost vylepšení a to je procházení posloupnosti střídavě shora a zdola. Tomuto postupu se říká *Shakesort* a je oproti původnímu algoritmu o hodně rychlejší.

Všechny popsané modifikace ovšem nemění počet potřebných záměn položek posloupnosti. Jediné co se změní, je počet porovnání hodnot.

3.2.2 Insertionsort – třídění přímým vkládáním

Tento algoritmus je opět velmi jednoduchý, pracuje tak, že vezme položku pole a vloží ji na její místo v posloupnosti. Provádí to tak, že vezme prvek posloupnosti a porovná ho vždy s jeho levým sousedem, pokud je tento větší, tak tohoto souseda posune o jednu pozici doprava. To dělá dokud nedosáhne začátku posloupnosti nebo dokud není soused menší. Poté vloží porovnávaný prvek na uvolněnou pozici [11].

Opět pracuje na místě, je stabilní a přirozený.

Program 3.2. Insertionsort

```
procedure InsertionSort;
var
  i, j: integer;
  x: TPrvek;
begin
  for i := 1 to length(seznam) - 1 do begin
    x := seznam[i];
```

```
      j := i - 1;
      while (j >= 0) and (seznam[j].klic > x.klic) do begin
        seznam[j + 1] := seznam[j];
        dec(j);
      end;
      seznam[j + 1] := x;
    end;
  end;
```

Analýza třídění vkládáním:

Algoritmus má nejlepší výsledky na předem částečně (nebo úplně) seřazených seznamech. Na náhodných datech nijak nevyniká, nicméně počet porovnání nutný k seřazení je o mnoho menší, než u bublinového třídění.

Postup lze výrazně vylepšit pokud použijeme nějakou metodu, která nám pomůže rychleji najít místo, kam prvek vložit. Uvědomíme-li si, že posloupnost, která vzniká je seřazená, můžeme použít k vyhledání správné pozice například metodu půlení intervalu. Tato metoda se jmenuje *binární vkládání*. Tento algoritmus je velice rychlý na posloupnosti, které jsou velmi neuspořádané na začátku, nebo ještě lépe na opačně seřazené posloupnosti, protože pro metodu půlení intervalu je jednodušší nalezení místa pro uložení prvku na levém konci posloupnosti, než na pravém. Toto je charakteristické pro nepřirozeně se chovající algoritmy.

3.2.3 Selectionsort – třídění přímým výběrem

Je to metoda, která je v principu opakem metody vkládání. Lze ji charakterizovat tím, že najde nejmenší prvek v seznamu, umístí ho na začátek posloupnosti a poté pracuje už jen s n - 1 prvky. Tento postup se opakuje, dokud se nedojde na konec posloupnosti [1].

Algoritmus je opět velmi jednoduchý, stabilní, pracuje na místě a je přirozený.

Program 3.3. Selectionsort

```
procedure SelectionSort;
var
  i, j, min: integer;
begin
  for i := 0 to length(seznam) - 1 do begin
    min := i;
    for j := i + 1 to length(seznam) - 1 do begin
      if seznam[min].klic > seznam[j].klic then begin
        min := j;
      end;
    end;
    Zamena(seznam[i], seznam[min]);
  end;
end;
```

Analýza třídění výběrem:

Počet porovnaných klíčů nezávisí na jejich uspořádání, takže z tohoto pohledu je metoda méně přirozená než řazení vkládáním. Nicméně počet porovnání neklesá ani nestoupá s ohledem na uspořádání posloupnosti. To je zaviněno i pevně daným počtem průchodů cykly. Z toho vyplývá i pevně daný počet přesunů.

3.2.4 Shellovo třídění – třídění vkládáním

Je to vylepšený algoritmus metody přímého vkládání. Navrhl ho v roce 1959 D. L.

Shell. Využívá toho, že řazení vkládáním je velmi efektivní, pokud je posloupnost předem alespoň částečně setříděna a že je neefektivní na hodnotách blízkých průměru, protože posouvá vždy jen o jednu pozici.

Shellovo třídění proto vylepšuje třídění vkládáním tak, že porovnává prvky, které jsou od sebe vzdálené o určitý krok. To umožňuje prvku konat rychlejší kroky k jeho výsledné pozici. Nad posloupností je provedeno několik průchodů, pokaždé s menším krokem, poslední krok je pak jednotkový a je to v podstatě čistá metoda třídění přímým vkládáním, ale s tím rozdílem, že data jsou již poměrně setříděná.

V porovnání s typem třídění, jako je například bublinové, u kterého trvá velmi dlouho, než se malá hodnota z konce posloupnosti dostane (po jednom kroku) na své místo, je Shellovo třídění efektivní v tom, že v prvních průchodech posloupností posouvá malé hodnoty z konce na začátek velkými kroky [1].

Metoda není stabilní a ani přirozenost metody není zcela jasná, nicméně pracuje v místě a je ze zatím uvedených algoritmů nejrychlejší.

Program 3.4. Shellsort

```
procedure ShellSort;
var
  i, j, krok: integer;
  x: TPrvek;
begin
  krok := length(seznam) div 2;
  while(krok > 0) do begin
    for i := krok to length(seznam) - 1 do begin
      j := i;
      x := seznam[i];
      while (j >= krok) and (seznam[j - krok].klic > x.klic) do
      begin
         seznam[j] := seznam[j - krok];
        j := j - krok;
      end;
      seznam[j] := x;
    end;
    krok := krok div 2;
  end;
end;
```

Analýza třídění podle Shella:

Algoritmus je zajímavý tím, že je velmi citlivý na velikost kroku. Obecně je doporučeno aby následující krok nebyl násobkem předchozího (což není v uvedeném příkladu dodrženo a krok je dělen dvěma). Každá sekvence dělení je sice funkční, protože dojde nakonec k jedničce, dosud ale není známo, jaká sekvence dělení je nejvýkonnější. Shell sám doporučil dělení n / 2 a dále, jako v příkladu, ale u některých jiných posloupností dělení je dosaženo ještě lepších výsledků.

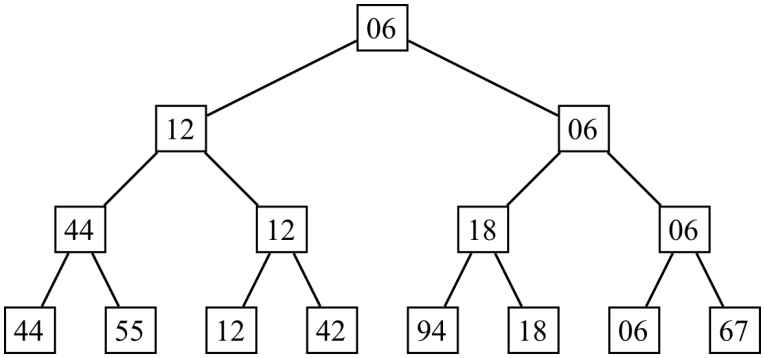
Podle Shellova dělení bylo dosaženo složitosti řádu $O(n^2)$. Jiným způsobem dělení lze dosáhnout až $O(n^{4/3})$ [10].

3.2.5 Heapsort – řazení haldou, stromové třídění

Třídění metodou výběru je založeno na opakujícím se výběru nejmenšího klíče z n prvků, potom $n - 1$ prvků atd. K vyhledání takovéhoto nejmenšího prvku je potřeba $n - 1$ porovnání z $n - 1$ prvků pak $n - 2$ porovnání atd. Ke zlepšení třídění metodou výběru je tedy

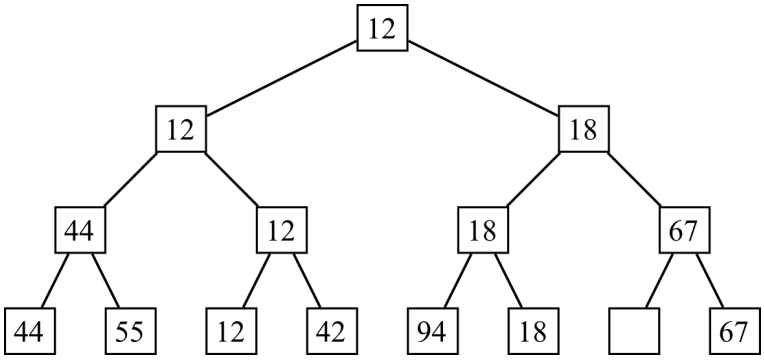
vhodné, získat větší množství informací z porovnávání v rámci každého průchodu, než je pouze informace o pozici nejmenšího prvku [1].

Například pomocí n / 2 porovnání se dá získat nejmenší klíč každé dvojice prvků, dalším n / 4 porovnáním se dá určit menší klíč z předchozích nalezených menších klíčů. Nakonec pomocí $n - 1$ porovnání můžeme sestrojit strom znázorněný na obrázku 3.1, jehož kořen obsahuje nejmenší prvek posloupnosti.



Obr. 3.1. Opakovaný výběr mezi dvěma klíči

V dalším kroku odstraníme nejmenší prvek z kořene stromu a současně tento prvek postupně ve stromě nahradíme buď prázdným vrcholem v případě listu, anebo odpovídajícím druhým (tedy větším) prvkem z každé porovnávané dvojice, ve které se prvek vyskytoval. Opakováním tohoto postupu získáváme v kořeni stromu druhý nejmenší prvek dané posloupnosti. Tento prvek vyloučíme stejným způsobem. Takto pokračujeme až do konce – tedy do té doby, dokud strom není prázdný.



Obr. 3.2. Zaplnění prázdných vrcholů

Můžeme říct, že každý krok výběru vyžaduje pouze $\log_2 n$ porovnání. A celkově je potřeba n kroků k sestrojení stromu a $n \log_2 n$ základních operací v rámci procesu výběru. To je výrazné zlepšení proti ostatním metodám, které principiálně vyžadují řád n^2 operací, dokonce i proti Shellovu třídění, které vyžaduje v nejlepším $n^{4/3}$ kroků.

Heapsort

Metodu stromového třídění používá metoda zvaná Heapsort – třídění haldou. Je to metoda, která zlepšuje stromové třídění.

Halda je definována jako posloupnost klíčů:

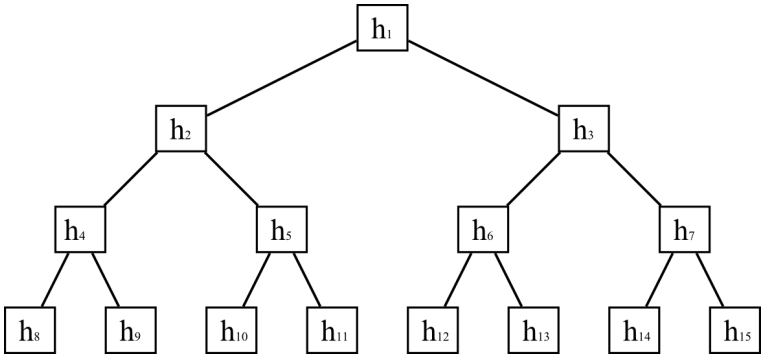
$h_l, h_{l+1}, \ldots, h_r$

takových, že platí:

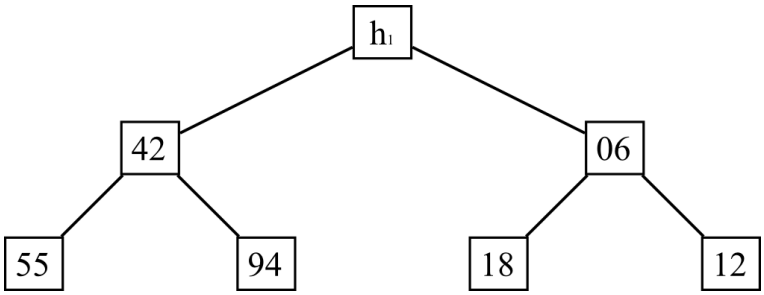
$h_i \leq h_{2i}$

$h_i \leq h_{2i+1}$

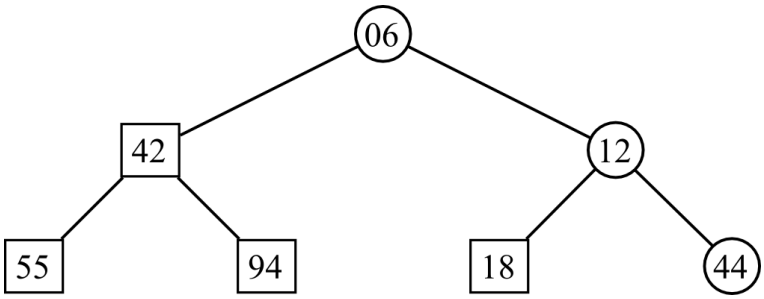
pro všechna $i = l, \ldots, r / 2$. Pokud je binární strom (vždy se dělí pouze na dvě větve) reprezentován polem, jako na obrázku 3.3, tak třídící stromy na obrázku 3.4 a 3.5 jsou haldy a platí: prvek h_1 je nejmenší prvek haldy $h_1 = \min(h_1, \ldots, h_n)$.



Obr. 3.3. Pole h zobrazené jako binární strom



Obr. 3.4. Halda se sedmi prvky



Obr. 3.5. Halda rozšířená o prvek s klíčem 44

Předpokládejme, že máme danou haldu s prvky $h_{l+1}, \ldots, h_r$ (pro nějaké l a r) a že do ní máme přidat nový prvek x tak, abychom dostali rozšířenou haldu $h_l, \ldots, h_r$. Jako příklad vezmeme haldu $h_1, \ldots, h_7$ znázorněnou na obrázku 3.4 a rozšíříme ji doleva o prvek $h_l = 44$.

Novou haldu sestrojíme takto: nejprve přiřadíme prvek x kořeni stromu a potom ho postupně porovnáme a vyměníme vždy s menším prvkem. V uvedeném příkladu je prvek s klíčem 44 nejprve porovnán s klíčem 6, potom 12 a nakonec vznikne halda zobrazená na

obrázku 3.5.

Nyní zformulujeme algoritmus tvorby haldy. Předpokládáme, že *l* a *r* jsou indexy označující prvky, které je třeba v rámci každého přechodu vyměnit. Jádrem algoritmu je procedura, která je uvedená jako program 3.5. Je dáno pole $h_1, \ldots, h_n$ a přitom prvky $h_{n/2+1}, \ldots, h_n$ už tvoří haldu, protože žádné dva indexy *l* a *r* nejsou takové, že by platilo $r = 2l$. Tyto prvky tvoří spodní řadu binárního stromu a nevyžadují žádné uspořádání.

Program 3.4. Tvorba haldy

```
procedure VytvorHaldu(l, r: integer);
var
  i, j: integer;
  x: TPrvek;
begin
  i := l;
  j := 2 * i;
  x := seznam[i];
  while j <= r do begin
    if j < r then
      if seznam[j].klic < seznam[j + 1].klic then
        j := j + 1;
    if x.klic >= seznam[j].klic then
      break;
    seznam[i] := seznam[j];
    i := j;
    j := 2 * i;
  end;
  seznam[i] := x;
end;
```

Halda se tak každým krokem rozšíří doleva o jeden nový prvek, který se prostřednictvím procedury VytvorHaldu dostane na správnou pozici.

Program 3.5. Třídění haldou s využitím procedury VytvorHaldu

```
procedure HeapSort;
var
  l, r, n:  integer;
  x: TPrvek;
begin
  n := length(seznam) - 1;
  l := (n div 2) + 1;
  r := n;

  while l > 0 do begin        //vytvoření haldy
    l := l - 1;
    VytvorHaldu(l, r);
  end;

  while r > 0 do begin        //setřídění
    Zamena(seznam[0], seznam[r]);
    r := r - 1;
    VytvorHaldu(l, r);
  end;

end;
```

Algoritmus tvorby haldy je vidět v prvním cyklu while programu 3.5. Pokud chceme získat prvky v uspořádaném tvaru, musíme vykonat *n* kroků vytvoření haldy, přičemž můžeme po každém kroku z kořene haldy vybrat následující prvek.

Poslední otázky, na které musíme najít odpovědi jsou: kde umístit jednotlivé prvky z kořene haldy a jestli je možné třídění na místě. Samozřejmě existují přijatelná řešení. V rámci každého kroku vybereme poslední prvek haldy a na jeho místo umístíme prvek z kořene haldy. Potom použijeme proceduru VytvorHaldu, pomocí které dostaneme prvek na své místo. Je na to potřeba *n*-1 kroků.

Toto je naznačeno v druhém cyklu while programu 3.5.

Analýza třídění haldou

Na první pohled není úplně jasné, jestli má tato metoda dobré výsledky. Je to proto, že nejprve se velké prvky přesouvají na začátek posloupnosti a až potom se začnou přesouvat na svoji správnou pozici. Proto se tato metoda nedoporučuje pro malé množství prvků *n*. Na velkém *n* je třídění haldou naopak velice efektivní. Čím větší *n* tím lepší výsledky. Dobré výsledky dostáváme i oproti Shellovu třídění. Nejhorší výsledek třídění dosažitelný tříděním haldou je $n \log_2 n$.

Není úplně jasné jaký vstup je pro třídění haldou nejlepší nebo nejhorší. Jeví se ale, že nejlepší výsledky dosahuje na zdrojových posloupnostech, které jsou částečně nebo úplně seřazené v opačném pořadí. Je to znakem nepřirozeně chovajícího se algoritmu. Třídění haldou se také chová nestabilně.

3.2.6 Quicksort – třídění rozdělováním, rychlé třídění

Tento algoritmus je považován za nejrychlejší algoritmus pro třídění polí. Princip algoritmu je založen na tom, že nejefektivnější výměny jsou na velké vzdálenosti.

Předpokládejme, že máme *n* prvků s obráceným pořadím klíčů. Tyto je možné setřídit pomocí *n* / 2 výměn. Nejprve se při tom porovnají prvky na levém a pravém konci posloupnosti a potom se postupuje z obou stran k opačnému konci a průchod se zakončí uprostřed. Toto samozřejmě platí, jen pokud víme, že pořadí prvků je opačné [1].

Pokusíme se tuto konstrukci zužitkovat pro vytvoření algoritmu, který by byl schopný realizovat zadané úkony:

- vybrat libovolný prvek *x* z tříděného pole
- prohledat prvky zleva, pokud se nenajde prvek $a_i > x$, potom prohledat prvky zprava, pokud se nenajde prvek $a_j < x$

Dále je potřeba tyto dva prvky vyměnit a pokračovat v procesu prohledávání a výměn, dokud se nesetkáme ve středu pole. Výsledkem tohoto je rozložení pole na dva úseky. V levém jsou všechny prvky menší než prvek *x* a v pravém úseku jsou všechny prvky větší, než prvek *x*. Naším cílem je ovšem seřazení pole. To je možné udělat rekurzivním rozdělením úseků na menší a menší až se dostaneme na jednotkové úseky.

Program 3.6. Quicksort

```
procedure QuickSort(l, r: integer);
var
  i, j: integer;
  x: TPrvek;
begin
 i := l;
 j := r;
 x := seznam[(l + r) div 2];
```

```
  repeat

    while seznam[i].klic < x do
      i := i + 1;

    while x < seznam[j].klic do
      j := j – 1;

    if i <= j then begin
      Zamena(seznam[i], seznam[j]);
      i := i + 1;
      j := j - 1;
    end;
  until i > j;

  if l < j then
    QuickSort(l, j);

  if i < r then
    QuickSort(i, r);
end;
```

Analýza třídění pomocí Quicksort

Program vybere prvek, takzvaný *pivot*, podle kterého rozdělí posloupnost na dvě přibližně stejné části (algoritmus také patří do třídy algoritmů rozděl a panuj). V těchto dvou částech pak přesune prvky menší než *pivot* nalevo a větší napravo. Po přeházení větších a menších prvků se algoritmus rekurzivně zavolá na jednu z rozdělených částí a potom na druhou.

Velmi důležitá je volba *pivota*, není nejvhodnější volit prvek z prostřední části pole, nebo náhodný prvek, ale prvek, který se blíží *mediánu* (hodnota, která je menší nebo rovna polovině prvků a větší nebo rovná druhé polovině prvků) dané posloupnosti. Výpočet *mediánu* je ovšem poměrně pomalá záležitost, protože je třeba projít celou posloupnost. Používá se tedy následujících metod:

- daná pozice posloupnosti – velmi nevýhodné, vzhledem k tomu, že nemáme přehled o ostatních prvcích. V programu 3.6 je použit prostřední prvek.
- náhodná pozice – poměrně používaná metoda, pokud je pozice pivota opravdu náhodná, běží algoritmus v $O(n \log_2 n)$ [9].
- výběr mediánu z více náhodných (nebo i fixních pozic), můžeme použít různý počet prvků.

Je dokázáno, že Quicksort je metoda nejrychlejší na náhodných vstupech, ovšem v krajních případech běží v řádu $O(n^2)$, což ji diskvalifikuje v některých aplikacích. Je to například tehdy, pokud je jako *pivot* vybrán pokaždé největší prvek úseku, což způsobí, že jeden podúsek bude mít jednotkovou velikost a druhý bude tak velký jako celá posloupnost bez jednoho prvku.

Algoritmus lze implementovat i nerekurzivně. Provede se to převedením rekurze na iteraci. Problém je v tom, že je potřeba uchovávat údaje o úsecích pole, protože vždy dělíme pole na dvě části, ale nemůžeme je zpracovat zároveň. Proto se používá zásobník, do kterého přidáváme údaje o začátku a konci úseku. Nerekurzivní verze je ve výsledku o něco rychlejší, protože nedochází ke stálému volání funkcí a má menší prostorovou náročnost.

3.3 Přehled

Na konec přehledu metod vnitřního třídění, musíme uvést jak jsou navzájem tyto metody efektivní.

3.3.1 Přehled metod z hlediska počtu operací

Na začátku jsme uvedli, že algoritmy je dobré posuzovat podle počtu porovnání (hodnota *C*) a přesunů (hodnota *M*) prvků, potřebných k setřídění posloupnosti. Testy byly prováděny na datech o velikosti 200 a 400 prvků. Data byla setříděná, náhodně uspořádaná a opačně setříděná.

Testy byly prováděny pomocí programu, který jsem naprogramoval v Borland Delphi 7, algoritmy jsou implementovány podle zde uvedených příkladů.

Tabulka 3.1. Počty přesunů a porovnání

Název		200 prvků			400 prvků		
		Setříděná	Náhodná	Opačná	Setříděná	Náhodná	Opačná
Bubblesort	C M	200 0	36400 9990	40200 20096	400 0	153600 38771	160400 80174
Bubblesort2*	C M	200 0	19472 9990	20300 20096	400 0	79187 38771	80600 80174
Insertionsort	C M	200 0	10174 10179	20100 20293	400 0	41171 41178	80198 80570
Selectionsort	C M	20100 201	20100 201	20100 201	80200 401	80200 401	80200 401
Shellsort	C M	1210 1210	2049 2149	1790 1982	2811 2811	5261 5465	4154 4545
Heapsort	C M	1492 1988	1427 1861	1330 1705	3407 4372	3265 4148	3074 3826
Quicksort	C M	1421 130	1961 422	1419 228	3242 266	4223 939	3240 460

* Vylepšená verze Bubblesortu, která nezachází do setříděných položek

Z tabulky je vidět, že jednoduché přímé metody třídění jsou velice efektivní na setříděné posloupnosti (naším cílem ovšem nebylo třídit setříděné), ale na neuspořádaných datech (i poměrně malého objemu) je počet operací, oproti složitějším metodám opravdu velký. Navíc počet operací roste exponenciálně, což není dobré.

Složitější metody jsou na tom o mnoho lépe, například Shellsort, i když je to jen vylepšený Selectionsort, vykazuje poměrně malé množství operací. Nejrychlejší je již podle názvu Quicksort, který má sice větší počet porovnání než Heapsort, ale má několikanásobně menší počet přesunů.

Při porovnání algoritmu samozřejmě také záleží na tom, která operace (přesun nebo porovnání) trvá na našem počítači déle. Obecně je to přesun prvků, který vyžaduje zápis do paměti. Porovnání je na dnešních strojích operace velice rychlá, je to vidět v tabulce 3.3, kde je měřena doba běhu algoritmu. Algoritmy, které na setříděné posloupnosti nevyžadují žádný přesun proběhnou téměř okamžitě.

3.3.2 Přehled podle vlastností algoritmů

Následuje srovnání podle teoretického řádu časové náročnosti algoritmů, stability a přirozenosti. Je vidět, že řád složitosti udává, jak rychle stoupá počet operací ve vztahu k velikosti vstupního souboru [4], [5].

Tabulka 3.2. Složitost algoritmů

Název	Časová složitost			Paměť	Stabilní	Přirozená	Metoda
	Min	Průměr	Max				
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)	ano	ano	záměna
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)	ano	ano	vkládání
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	ano	ano	výběr
Shellsort	?	?	$O(n^{4/3})$	O(1)	ne	ano?	vkládání
Heapsort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	O(1)	ne	ne	výběr
Quicksort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	O(1)*	ne	ne	záměna

* záleží na implementaci, určité verze vyžadují dodatečnou paměť

Pro Shellovo třídění zatím nebyly řády složitosti nalezeny a ani přirozenost algoritmu není jistá a různé zdroje se v tomto údaji liší.

Je vidět, že použitím rychlých metod třídění se připravíme o stabilitu algoritmu. Pokud je tedy posloupnost tříděná podle nějakého sekundárního klíče, je potřeba ji po provedení třídění setřídit ještě podle něj (pokud nepoužíváme komplexní porovnávací funkci), což může celkový proces třídění zpomalit, ale proti použití přímých metod to není nijak bolestné, zvlášť na velkém souboru, kde ušetřený čas roste exponenciálně s počtem prvků.

3.3.3 Porovnání podle doby trvání algoritmu

Další užitečné porovnání je podle času potřebného k setřídění nějaké posloupnosti v závislosti na velikosti a tvaru vstupních dat. Porovnání jsem provedl na setříděných, opačně setříděných a náhodně uspořádaných vstupních datech, vstupní data měly velikost 5 000 a 10 000 prvků.

Testy byly prováděny pomocí programu, který je zmíněn v § 3.3.1. Programu byla předělena priorita „realtime“, všechny testy byly provedeny na stejném stroji (AMD Athon 1133Mhz, 512MB RAM). Výsledky jsou uváděny v milisekundách.

Výsledky se liší od analytických řádů složitosti z tabulky 3.2. Je to proto, že například pro Quicksort není nejhorší možností opačně setříděná posloupnost, ale případ, kdy je vybrán špatně pivot. Výběr pivota se ovšem dá ošetřit programově, daleko pravděpodobnější je, že dostaneme opačně setříděnou posloupnost.

Výsledky jsou samozřejmě jen orientační, a na jiných datech se budou lišit.

Tabulka 3.3. Časová náročnost v milisekundách

Název	5 000 prvků			10 000 prvků		
	Setříděná	Náhodná	Opačná	Setříděná	Náhodná	Opačná
Bubblesort	5	5388	10395	5	22172	43273
Bubblesort2*	5	5248	10245	5	21391	42200
Insertionsort	5	2133	4436	5	8963	18717
Selectionsort	120	120	150	460	470	610
Shellsort	30	40	35	70	100	80
Heapsort	30	30	25	65	65	60
Quicksort	5	17	5	15	35	20

* Vylepšená verze Bubblesortu, která nezachází do setříděných položek

Pro zajímavost jsem pořídil ještě výsledky Shellsort, Heapsort a Quicksort pro 100 000 prvků.

Tabulka 3.4. Rychlejší metody na vetším objemu dat

Název	100 000 prvků		
	Setříděná	Náhodná	Opačná
Shellsort	1172	1893	1372
Heapsort	832	1000	811
Quicksort	245	531	300

Z tabulek je vidět propastný rozdíl mezi přímými a složitějšími metodami. To co Quicksort zvládne během pár milisekund, provádí Bubblesort dokonce několik desítek sekund. Je tedy jasné, že pro jakékoliv větší soubory dat jsou přímé metody nepoužitelné. Jejich nasazením do takovýchto podmínek bychom náš program naprosto nesmyslně zpomalili a to by jistě nepotěšilo uživatele našeho softwaru.

4 VNĚJŠÍ ALGORITMY TŘÍDĚNÍ

Tento druh algoritmů již v dnešní době nemá příliš velké opodstatnění, jelikož dnešní velikosti operačních pamětí dosahují několika gigabajtů. Navíc je možnost rozdělit celý proces třídění na více počítačů (například Quicksort se dá poměrně lehce paralelizovat). Jediný problém je potom dopravit data na jiný počítač a tam je roztřídit a zaslat zpět.

V dnešní době navíc pro ukládání dat používáme vyspělé databázové systémy, do kterých se data ukládají průběžně a pokud je z nich potřeba nějaký výstup, stačí zaslat na databázový systém dotaz například v SQL jazyku (Structured Query Language – strukturovaný dotazovací jazyk) a data dostaneme zpět přesně v takové formě jaká je požadována na straně klienta.

Pokud jde o dnes populární relační databáze (pohlížejí na data jako na tabulky), většinou využívají takzvané indexy. Jsou to vybrané hodnoty, které databázový program udržuje stále setříděné mimo vlastní data a je k nim přiložena informace, kde se nacházejí data, která k danému klíči patří. Indexy umožňují velice rychlý přístup k uloženým položkám. Indexovaná data jsou určena při návrhu databáze s ohledem na to, jaká data a v jaké formě budeme během používání aplikace spolupracující s databází požadovat. Vyspělejší systémy mohou indexy generovat podle toho, jaké jsou na ně kladeny požadavky.

Pokud do databáze přibude nějaká položka, je obvykle zapsána jako další prvek do souboru uchovávaných dat, ale klíč, podle kterého se data indexují, je přidán na správné místo indexované posloupnosti – to může být provedeno velmi rychle, protože posloupnost indexů je udržována setříděná.

Vnější třídění

V minulosti byly uvedené metody vnitřního třídění nepoužitelné, pokud se soubor dat nevešel do operační paměti počítače. Paralelizace procesu nebyla možná z toho důvodu, že počítače zabíraly několik místností a byly velmi drahé, jak na pořízení, tak na provoz, proto nebylo možné jich mít více najednou. A pokud byly propojeny, tak rychlost přenosu dat nebyla nijak závratná. Ostatně ani počítače nebyly nikterak rychlé proti dnešním strojům. V té době se obyčejně pracovalo se sekvenčními paměťovými zařízeními jako jsou například magnetické pásky.

Charakteristickou vlastností sekvenčních souborů je, že v každém okamžiku je k dispozici pouze jeden prvek souboru. Vzhledem k tomuto silnému omezení je potřeba používat jiné techniky třídění než u třídění polí.

Nejpoužívanější je technika přímého slučování [1]. Znamená to spojování dvou a nebo více posloupností do jedné setříděné posloupnosti prostřednictvím výběru mezi momentálně přístupnými prvky. Slučování je podstatně jednodušší a používá se jako pomocná operace pro složitější proces třídění sekvenčních souborů. Jedním z nejjednodušších algoritmů vnějšího třídění slučováním je přímé slučování, které pracuje takto:

1. posloupnost *A* rozdělíme na dvě poloviny *B* a *C*
2. spojováním prvků do uspořádaných dvojic se posloupnosti *B* a *C* sloučí do nové posloupnosti *A*'
3. na posloupnost *A*' se aplikují kroky 1 a 2, čímž dostaneme uspořádané čtveřice
4. první tři kroky se postupně opakují, tím získáváme uspořádané osmice atd., pokaždé se zdvojnásobí délka slučovaných posloupností. Algoritmus končí tehdy, když je celá posloupnost uspořádaná.

Takovýto algoritmus potřeboval pro svou činnost tři záznamové pásky. Vzhledem k tomu, že fáze rozdělení dat nepřináší žádné přeuspořádání dat, je tato fáze neproduktivní. Navíc tato fáze představuje téměř polovinu všech kopírovacích operací. Snahou je tedy tuto

fázi odstranit, což se dá lehce udělat tak, že spojíme rozdělování a slučování do jedné fáze. Namísto slučování prvků do jedné posloupnosti, se budou prvky rovnoměrně rozdělovat na dvě pásky. Je tím odstraněna polovina kopírovacích operací, čímž dochází k výraznému zrychlení. Nicméně je potřeba ještě čtvrtá páska.

5 ZÁVĚR

V práci jsem definoval problém třídění dat. Popsal vybrané algoritmy třídění a na závěr jsem zhodnotil jejich efektivitu a časovou náročnost.

K tomuto účelu jsem zhotovil počítačový program v Delphi. Tento program slouží nejen k vizualizaci vybraných algoritmů, ale také zobrazuje počet přesunů, porovnání a délku celého procesu třídění, která ovšem při malém počtu položek, se kterými se pracuje, není příliš vypovídající. Pro měření s velkým počtem položek bylo zapotřebí program překompilovat, tak aby počítal s opravdu velkým počtem položek. V běžícím programu je možno pouze nastavit jakou posloupnost vygenerovat (setříděná, náhodně uspořádaná, opačně setříděná), následně vybrat algoritmus a spustit třídění.

Při tvorbě programu a následném měření jsem zjistil, že v dnešní době není do určitého množství prvků v tříděném seznamu (asi do tisíce prvků) velký rozdíl mezi jednotlivými algoritmy, jejich rychlost nebo pomalost je sice nezpochybnitelná, ale na rychlém počítači člověk třídící proces ani není schopen zaznamenat. Samozřejmě, pokud je třídění jediný úkon, který by náš program konal a dělal by jej s velkou frekvencí, je dobré použít některý z vyspělejších postupů třídění. Na malém množství dat a pokud není funkce třídění příliš vytěžovaná, se není potřeba trápit s implementací nějaké složitější metody.

Obrovských rozdílů ale dosáhneme, pokud třídíme posloupnost, která má počet prvků v řádů desetitisíců. Na takovýchto posloupnostech jsou přímé metody nepoužitelné a jejich použití se rovná mrhání strojovým časem.

V přehledu je zmíněno pouze několik nejzákladnějších algoritmů, každý je ale nějakým způsobem jedinečný ať už je to jednoduchost nebo rychlost, ale hlavně pracuje každý odlišným způsobem. Za dobu, po kterou používáme počítače vzniklo několik desítek algoritmů pro třídění posloupností. Většina z nich pak vychází právě ze zmíněných několika základních postupů, například existuje postup, který kombinuje Quicksort a po několika jeho průchodech přejde na Heapsort, kterým operaci dokončí.

Existují ještě algoritmy, které vyžadují ke své činnosti dodatečnou paměť nebo dokonce speciální hardware. Takovéto algoritmy jsou ovšem ještě složitější pro implementaci. Jejich výhodou je ovšem rychlost větší než u algoritmů pracujících na místě.

6 SEZNAM POUŽITÉ LITERATURY

[1] Wirth, N. Algoritmy a štruktúry údajov. 2. vyd. Bratislava : Alfa, 1989. 481 s. Edícia výpočtovej techniky. ISBN 80-05-00153-3

[2] Honzík, J. Programovací techniky. 1. vyd. Brno : VUT Brno, 1985. 357 s. ISBN 55-588-85

[3] Wikipedia, Algortimus [online]. 10. 4. 2007. Dostupné z <http://cs.wikipedia.org/wiki/Algoritmus>

[4] Wikipedia, Algoritmus řazení [online]. 22.4.2007. Dostupné z <http://cs.wikipedia.org/wiki/Algoritmus_%C5%99azen%C3%AD>

[5] Wikipedia, Sorting algorithm [online]. 12.5.2007. Dostupné z <http://en.wikipedia.org/wiki/Sorting_algorithms>

[6] Wikipedia, Rekurze [online]. 7.5.2007. Dostupné z <http://cs.wikipedia.org/wiki/Rekurze>

[7] Wikipedia, Pole [online], 4.5.2007. Dostupné z <http://cs.wikipedia.org/wiki/Pole_%28informatika%29>

[8] Wikipedia, Buble sort [online], 18.4.2007. Dostupné z <http://cs.wikipedia.org/wiki/Bubble_sort>

[9] Wikipedia, Quicksort [online], 2.5.2007. Dostupné z <http://cs.wikipedia.org/wiki/Quicksort>

[10] Wikipedia, Shell sort [onlline], 10.5.2007. Dostupné z <http://en.wikipedia.org/wiki/Shell_sort>

[11] Wikipedia, Insertion sort [online], 14.5.2007. Dostupné z <http://en.wikipedia.org/wiki/Insertion_sort>

[12] Manualy.net, Algoritmy: složitost [online], 4.5.2006. Dostupné z <http://www.manualy.net/article.php?articleID=14>

[13] Beran, R. Algoritmy [online]. Dostupné z <http://www.beranr.webzdarma.cz/algoritmy/algoritmy.html>

[14] KSP, Třídění [online], 7.3.2007. Dostupné z <http://ksp.mff.cuni.cz/tasks/18/cook1.html>

[15] Wikipedia, Pseudocode, 15.5.2007. Dostupné z <http://en.wikipedia.org/wiki/Pseudocode>