Univerzita Hradec Králové

Přírodovědecká fakulta

Katedra informatiky

Třídící algoritmy

Bakalářská práce

Autor: Martin Tulis

Studijní program: B1101 Matematika

Studijní obor: Matematika se zaměřením na vzdělávání
Informatika se zaměřením na vzdělávání

Vedoucí práce: Ing. Jiří Jelínek, Ph.D.

Hradec Králové duben 2015

Prohlášení:

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a že jsem v seznamu použité literatury uvedl všechny prameny, ze kterých jsem vycházel.

V Hradci Králové dne 27. 4. 2015 Martin Tulis

Poděkování

Děkuji vedoucímu práce Ing. Jiřímu Jelínkovi, Ph.D. za cenné rady, připomínky a čas, který mi věnoval při konzultacích a vedení práce.

Anotace

Tulis, M. *Třídící algoritmy*. Hradec Králové, 2015. Bakalářská práce na Přírodovědecké fakultě Univerzity Hradec Králové. Vedoucí práce Jiří Jelínek. 71 s.

Práce se zabývá řazením a třídícími algoritmy, jejich popisem, porovnáním a implementací do jednoduchých programů. V první části se věnuje úvodu do tématu programování, algoritmizace a představením programovacího jazyku JAVA. Ve druhé části detailně popisuje jednotlivé řadící algoritmy, jejich princip, vlastnosti a náročnost. V praktické části práce jsou jednotlivé řadící algoritmy naprogramované a jejich činnost a náročnost je ukázána na příkladech řazení náhodných čísel. Součástí praktické části je i vyhodnocení didaktické vhodnosti k výuce jednotlivých algoritmů. V závěru práce jsou algoritmy porovnány a výsledky vyhodnoceny.

Klíčová slova

Třídění, algoritmy, řazení, JAVA, programování, bubblesort, selectsort, insertsort, quicksort, mergesort, combsort, shellsort, heapsort

Anotation

Tulis, M. *Sorting algorithms*. Hradec Králové, 2015. Bachelor thesis at Faculty of Sciece University of Hradec Králové. Thesis Supervisor Jiří Jelínek. 71 p.

The work deals with the order and sorting algorithms, their description, comparison and implementation in simple programs. The first part is devoted to an introduction to the topic of programming, algorithms and performance Java programming language. The second part describes in detail the various sorting algorithms, their principles, characteristics and demands. In the practical part are different sorting algorithms programmed and their activities and demands exemplified sort of random numbers. The practical part of the teaching and assessment of suitability for teaching the algorithms. In conclusion, the algorithms are compared and the results evaluated.

Key words:

Sorting algorithms, JAVA, programming, bubblesort, selectsort, insertsort, quicksort, mergesort, combsort, shellsort, heapsort

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Třídění a seřazovaní dat do posloupností podle zvoleného kritéria je problém, který lidé řeší od nepaměti. Kritéria třídění se mohou lišit podle potřeby, ale také podle druhu vstupních dat. Číselná data jsme zvyklí řadit podle velikosti buď v sestupném, nebo vzestupném pořadí. U jmenných seznamů budeme volit třídění dle abecedy. V jiném případě například podle délky řetězce. Volba je vždy závislá na způsobu následného použití dat.

Rozdíl je mezi možnostmi jak data nejrychleji setřídit. Zatímco si dříve lidé museli vystačit sami, od poloviny 20. století získali silný a rychlý nástroj - počítač. Stroj, který měl potenciál provádět třídění rozsáhlých dat snadno a rychle. To se stalo impulsem pro inženýry a matematiky vymyslet, jak počítače pro třídění dat využít. Tím se začaly rodit třídící algoritmy - postupy jak vstupní data setřídit podle požadovaného kritéria a pořadí.

Dnes a denně se setkáváme se setříděnými daty, považujeme je za samozřejmost a ani si jejich přítomnost neuvědomujeme. Nikoho nepřekvapí, že se v telefonním seznamu kontaktů, jednotlivá jména řadí podle abecedy, zprávy v emailovém klientovy podle data doručení, studenti u přijímacích zkoušek podle dosažené úspěšnosti, atleti podle zdolané vzdálenosti nebo dosaženého času, žáci podle data narození, pacienti podle rodného čísla. To vše jsou seznamy založené na svém pořadí. Důvod potřeby lidí data třídit je patrný. Setříděný seznam velmi urychluje základní operace - vyhledávání a porovnávání jednotlivých záznamů.

Cílem bakalářské práce je popsat princip třídících algoritmů a prakticky jednotlivé třídící algoritmy porovnat na různých vstupních datech. Teoretická část se věnuje úvodu do programování a definici základních pojmů algoritmus a třídění. Dále předkládá způsoby klasifikace třídících algoritmů a podrobně jednotlivé algoritmy popisuje včetně rozboru vlastností a implementace. Praktická část na vzorových vstupních datech ověřuje vlastnosti jednotlivých algoritmů a celkovou časovou náročnost. Z důvodu častého zařazování třídících algoritmů do výuky středních odborných škol je v práci obsažena kapitola popisující didaktickou vhodnost jednotlivých algoritmů pro začlenění do středoškolské výuky.

1 Základní pojmy programování v jazyce JAVA

1.1 Syntaxe

Úvodem uveďme, že programovací jazyk JAVA je case sensitive, to znamená že rozlišuje malá a velká písmena, navíc podporuje celou sadu UNICODE. Při tvorbě programu v JAVA využíváme těchto syntaktických elementů:

- Prázdná místa
- Komentáře
- Klíčová slova
- Identifikátory
- Literály
- Separátory, oddělovače
- Operátory

Prázdná místa (někdy označovány jako bílá místa) jsou všechny mezery, tabulátory a znaky konců řádek. V místě prázdného místa lze vložit komentář.

Komentáře nejsou překládány, díky tomu můžeme při jejich vkládání využívat všechny znaky sady Unicode. Jednotlivé komentáře jsou od programu odděleny specifickými znaky. Jednořádkový komentář je oddělen znakem // a je ukončen až koncem řádku, víceřádkový komentář vkládáme mezi znaky /* */ a dokumentační mezi znaky /** */.
Obsahem komentáře by měl být popis principu programu nebo jeho klíčových částí. Také je vhodné komentář připojovat při deklaraci konstant a proměnných.

Klíčová slova jsou vyhrazené identifikátory pro konstrukční prvky programu (např. zápis příkazu). Klíčová slova se vždy píší malými písmeny a jsou pevně daná.

```
abstract  assert  boolean  do  break  byte case catch char class
const  continue  default  double  else  enum  extends  final
finally  float  for  goto  if  implements  import  instanceof
int  interface  long  native  new  package  private  protected
public  return  short  static  strictfp  super  switch
synchronized this throw throws transient try void volatile while
```

Existují ještě další rezervovaná slova, která se někdy mezi klíčová slova zařazují. Jsou to hodnoty (literály) typu boolean: `true`, `false` a `null`.

Identifikátor je jednoznačný název, kterým pojmenováváme třídy, proměnné, metody tříd, konstanty, balíčky. Pro volbu identifikátorů platí několik podmínek:

- skládá se z písmen anglické abecedy, číslic a znaku podtržítko „_“
- nesmí začínat číslicí
- nelze využívat klíčová slova
- znak tečka „.“ odděluje jednotlivé části složených identifikátorů

Kromě uvedených podmínek jsou programátory respektována další pravidla, která zvyšují přehlednost programu:

- identifikátory volíme krátké a výstižné (některé překladače respektují pouze prvních 32 znaků)
- třídy a rozhraní – velká počáteční písmena

```
StringBuffer
```

- proměnné a metody – malé počáteční písmeno

```
pocetCisel
```

- konstanty – pouze velká písmena, lze využívat podtržítko „_“

```
MIN_INDEX
```

- balíčky – pouze malá písmena

```
java.lang
```

Separátory oddělují jednotlivé části programu a příkazy. Jazyk JAVA využívá tyto závorky a interpunkční znaménka.

```
( )   [ ]   { }   ;   ,   .
```

Operátory umožňují práci s jednotlivými operandy. Nejběžnějším operátorem je přiřazení „=“, který se používá při každém přiřazení hodnoty do proměnné. Použitím operátoru přiřazení vzniká přiřazovací příkaz.

Základní přehled nejběžnějších operátorů si ukážeme v tabulce:

Aritmetické operátory		Logické operátory	
+	součet	==	rovnost
-	rozdíl	!=	nerovnost
*	součin	&&	AND log. součin
/	podíl	||	OR log. součet
%	celočíselný zbytek	!	NOT negace
++	inkrementace	< >	větší, menší
--	dekrementace	>= <=	větší, menší nebo rovno

Tabulka 1: Přehled operátorů

Další potřebné operátory můžeme získat pomocí importování tříd. Například operace matematického charakteru můžeme používat pomocí třídy Math, která přidává mocniny, odmocniny atd.

Zvláštním typem je ternární operátor, který umožňuje vytvořit **podmíněný výraz**.

```
//podmíněný výraz
podminka ? vyraz1 : vyraz2

//příklad
cislo = cislo < 0 ? 0 : 1
```

Je-li podmínka splněna, tedy proměnná `cislo` je menší než nula, pak se do proměnné `cislo` přiřadí hodnota 0, v opačném případě se do proměnné přiřadí hodnota jedna. (Schildt, 2001)

1.2 Proměnné

Proměnná je pojmenované místo v paměti, kde máme uloženou hodnotu. Použitím proměnné se odkazuji právě na toto místo v paměti a využívám uloženou hodnotu.

Primitivní datové typy jazyku JAVA

Primitivních datových můžeme rozdělit na celočíselné, reálné, znakové a logické datové typy.

Základním **celočíselným** datovým typem je `int`, jeho rozšířením je typ `long`, naopak zmenšený typ je `short` a `byte`. Tyto typy respektují znaménko minus „-“ a lze tedy zadávat i záporné hodnoty. Navíc můžeme čísla zadávat i v soustavě osmičkové (oktalové) a šestnáctkové (hexadecimální). Není k tomu zapotřebí žádná zvláštní definice, jedná se pouze o způsob zapsání hodnoty. Začíná-li hodnota nulou, překladač ji považuje za hodnotu v osmičkové soustavě. Podobně, pokud hodnota začíná „0x“, pak je hodnota zapsaná v hexadecimální soustavě.

Pro čísla s desetinou čárkou využíváme **reálné datové typy** `float` a `double`. Tyto typy jsou pro práci s desetinou čárkou uzpůsobené, proto můžeme i používat zápis s řádem vyjádřeným pomocí exponentu. V případě potřeby navíc můžeme použít i hodnotu nekonečno.

```
float f = 5.013;
double d = .125e-2;          //=0,00125
double n = POSITIVE_INFINITY; //+nekonecno
```

Zvláštním datovým typem je `char`. Tento **znakový datový typ** má jako svoji hodnotu znak sady UNICODE. Zápis provedeme přiřazením šestnáctkového kódu znaku, nebo samotného znaku v apostrofech (pozor, ne uvozovky).

```
// shodny zapis
char a = 'a';
char b = /u00C1;
```

`Boolean` je jediným **logickým datovým typem**. Jeho velikost je jeden bit a nabývá tedy pouze dvou hodnot, `true` (pravda) a `false` (nepravda).

Primitivní datové typy	velikost v bytech	rozsah	implicitní hodnota
boolean	1 bit	true, false	false
char	2	65 536 různých znaků	\u0000
byte	1	−128 až +127	0
short	2	−32 768 až +32 767	0
int	4	−2 147 483 648 až +2 147 483 647	0
float	4	±3.402 823 47 E+38	0.0
long	8	−9 223 372 036 854 775 808 až +9 223 372 036 854 775 807	0
double	8	±1.797 693 134 862 315 70 E+308	0.0

Tabulka 2: Přehled primitivních datových typů

Deklarace a inicializace

Základem pro práci s proměnnými je jejich deklarace (vytvoření) a inicializace (přiřazení hodnoty). Jazyk JAVA je přísně typový jazyk, to znamená, že každá proměnná má svůj identifikátor (tj. jméno) a pevně určený datový typ. Deklarace se v Javě provede přiřazením datového typu, poté proměnná může obsahovat pouze hodnotu daného typu. V praxi lze využít několik způsobů jak deklaraci a inicializaci použít, pro přehlednost kódu jsou však doporučené jen některé.

Různé způsoby deklarace i inicializace si ukážeme na následujících příkladech:

```
int i;  //pouze deklarace
i = 1500; //inicializace

// doporuceny postup
int j = 1500; //deklarace s inicializaci

// nedoporuceny postup
int a, b, c; //deklarace vice promennych
int a = 7, b = c, d = 17;
```

Hodnota nemusí být nutně zadána pouze konstantou (hodnotou), ale lze použít i výraz. Podmínkou je, že výraz musí být v okamžiku překladu vyhodnotitelný.

```
//priklad
long l = 30 + 60;

//priklad
int a = 20;
int b = a * 2;
```

Neurčí-li programátor při deklaraci proměnné její hodnotu, kompilátor přiřadí implicitní hodnotu automaticky. Je ovšem programátorským zvykem tyto hodnoty k proměnným inicializovat. Není přesně určeno, jak se mají překladače zachovat při vložení počáteční hodnoty mimo rozsah datového typu proměnné (např. `byte b = 500`). Většina ohlásí chybu, ale některé překladače přiřadí chybnou hodnotu. (Keogh, a další, 2006)

1.3 Příkazy

Příkazy jsou jednotlivé kroky programu. V JAVA každý příkaz ukončujeme středníkem a je zvykem jej psát na samostatný řádek. Základními typy příkazů jsou tzv. **výrazové příkazy**, mezi které zařazujeme přiřazovací výrazy, příkazy inkrementace a dekrementace, volání metod a příkaz vytvoření nového objektu.

Další skupinou jsou **deklarativní příkazy**, které jsme si představili v předchozí kapitole věnující se tvorbě proměnných. Poslední velkou skupinou jsou příkazy určené pro **řízení toku programu**.

Prázdný příkaz

Prázdný příkaz se příliš nevyužívá, přestože ojediněle jej můžeme v programu nalézt. Prázdný příkaz je tvořený samostatným středníkem umístěným ve zdrojovém kódu.

Blok

Blok je skupina příkazů uzavřená ve složených závorkách „{ }“. Celý blok příkazů se překladači jeví jako jeden příkaz a proto jej můžeme využít v místech, kde nám syntaxe umožňuje vložit příkaz jen jeden.

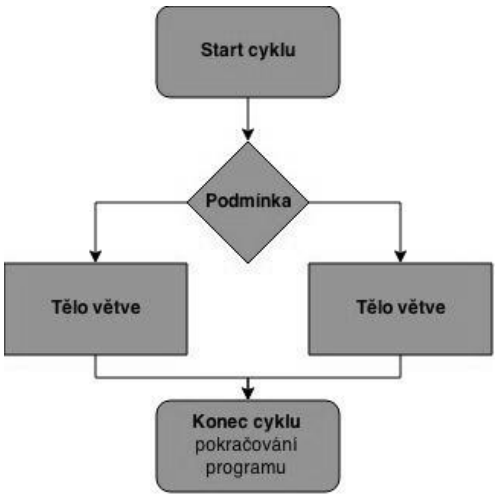
Součástí bloku může být další (vnořený) blok. Pro přehlednost kódu by měla být otevírací i zavírací závorka na samostatném řádku. Všechny příkazy v bloku jsou poté zleva odsazeny. (Prokop, 2009)

1.4 Větvení programu

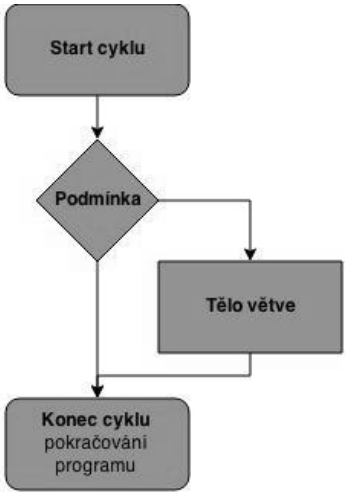
IF, ELSE

Příkaz `if` (respektive jeho rozšíření `if-else`) nazýváme podmíněný příkaz a setkáme se s ním velmi často. Tento příkaz rozhoduje o dalším průběhu programu na základě hodnoty logického výrazu (podmínky).

```
if (a > 0)                          // je-li a větší než 0
  System.out.print(„a je kladné“)   // vypiš „a je kladné“
  else                             // neplati-li že a je větší než 0
  System.out.print(„a je záporné“) // vypiš „a je záporné“

//zkrácená varianta
  if (podminka)
  prikaz;
```



Obrázek 1: Vývojový diagram příkaz If



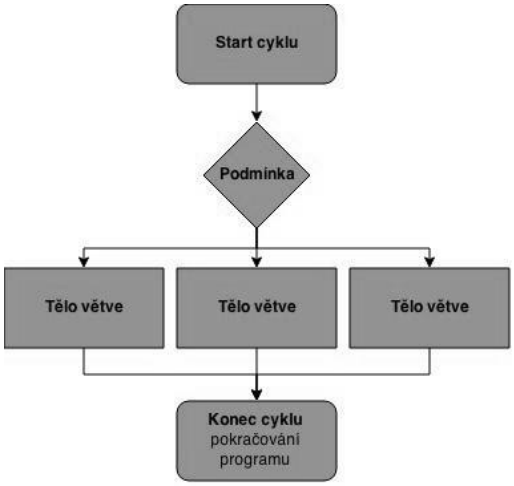
Obrázek 2: Vývojový diagram příkaz If-else

Možnou variantou je i zkrácení tohoto příkazu vynecháním bloku `else`. Při nesplnění podmínky program pokračuje dalšími příkazy za blokem `if`.

Příkaz SWITCH

Tento příkaz nám plní funkci přepínače. Na rozdíl od příkazu `if-else` dokaže nabídnout více než dvě možnosti a tak nám dovolí program vícenásobně větvit.

```
switch (vyraz)
{
  case 0:   //hodnota vyrazu
    System.out.println(„nula“);
    break;
  case 1:
    System.out.println(„jedna“);
    break;
  case 20:
    System.out.println(„dvacet“);
    break;
  default:
    System.out.println(„jiná hodnota“);
    break;
}
```



Obrázek 3: Vývojový diagram příkaz switch

Pomocí hodnoty proměnné `vyraz` volíme mezi jednotlivými větvemi programu. Při hodnotě `vyraz = 0` se bude provádět blok `case` 0. Zadá-li uživatel do proměnné výraz jiné hodnoty než námi uvedené (tj. 0,1,20), provede se blok `default`.

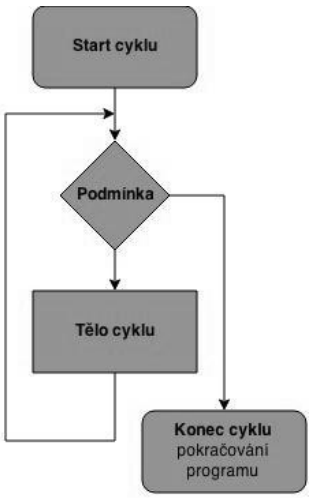
Výraz `break` znamená okamžité ukončení cyklu. Program bude pokračovat následujícím řádkem za cyklem. Přestože není příkaz `break` povinný, pro správnou funkci přepínače jej nesmíme zapomenout. (Kadlec, 2002)

1.5 Cyklické příkazy

Cyklus WHILE

Cyklus s podmínkou na začátku. Tento cyklus používáme v případě, kdy neznáme přesný počet iterací a zároveň může nastat situace, kdy cyklus neproběhne ani jednou.

```
while (podminka)
{
  prikaz1;
  prikaz2;
}
```



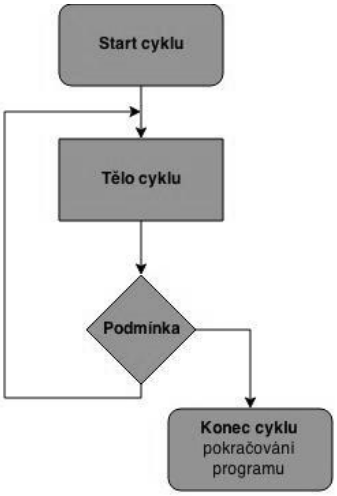
Obrázek 4: Vývojový diagram cyklu while

Výraz (podmínka) se testuje ještě před začátkem cyklu. Jednotlivé iterace cyklu se opakují, dokud splňují podmínku, v případě nesplnění podmínky se cyklus neprovádí a program pokračuje dalším řádkem.

Cyklus DO-WHILE

Cyklus s podmínkou na konci. Používáme jej v případech, kdy je potřeba alespoň jeden průběh a zároveň neznáme přesný počet iterací.

```
do{
  prikaz1;
  prikaz2;
}
while (podminka);
```



Obrázek 5: Vývojový diagram cyklus do-while

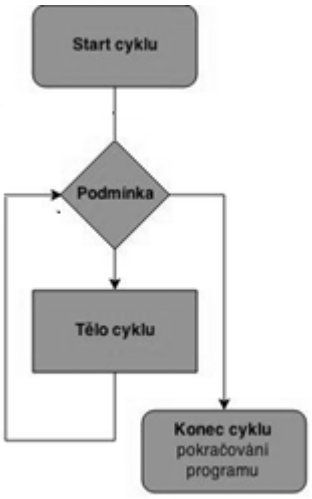
Podmínka se ověřuje až po prvním průběhu cyklu, je-li podmínka splněna, cyklus pokračuje další iterací. V opačném případě již další iterace neprobíhá a program pokračuje dalším řádkem po cyklu.

Cyklus FOR

Cyklus se známým počtem iterací. Za klíčovým slovem se v kulatých závorkách nachází hned trojice výrazů. První je inicializace počáteční hodnoty řídící proměnné, druhý výraz je podmínka. Posledním výrazem je změna hodnoty řídící proměnné, která se provede vždy po provedení cyklu. Jednotlivé výrazy jsou od sebe odděleny středníkem.

```
for (inicializace; podminka; zmena;){
 prikaz1;
 prikaz2;
}
```

```
for (int i = 1; i < 10, i++){
 prikaz1;
 prikaz2;
}
```



Obrázek 6:Vývojový diagram cyklu for

Před každým průběhem cyklu se ověří podmínka, je-li podmínka splněna, cyklus probíhá a po jeho ukončení dochází ke změně hodnoty řídící proměnné. (Prokop, 2009)

2 Algoritmy

2.1 Definice pojmu algoritmus

Původ slova algoritmus můžeme nalézt v Persii. Historici se domnívají, že bylo odvozeno z latinského přepisu jména perského matematika, astronoma a geografa Abu Ja'far Muhammad ibn Musa al-Khwarizmi. Al-Khwarizmi žil a tvořil v letech asi 780 – 850 n.l. a jeho náplní bádání byla především algebra. Jeden z jeho životních spisů byl ve 12. století latinsky přeložen pod názvem *Algoritmi de numero Indorum* (Algoritmi o číslech od Indů) a právě autorovo jméno Algoritmi se stalo základem slova algoritmus, které se zpočátku používalo pro popis matematických postupů a asi od 20. století se termín používá v dnešním významu.

Milková **definuje algoritmus** jako *„postup skládající se z jednotlivých jednoznačně určených kroků, tzv. příkazů. Každý algoritmus by měl dodržovat následující principy: hromadnost determinovanost, konečnost, rezultativnost*“ (Milková, 2008 str. 8)

V knize *Umění programování* (Knuth, 2008) se vlastnosti algoritmu ještě doplňují o efektivitu.

Uvedené vlastnosti můžeme popsat následovně. Algoritmus řeší celou třídu konkrétních problémů, lišících se pouze vstupními údaji (**hromadnost**). Jednotlivé kroky algoritmu i jejich návaznost musí být jednoznačně určena (**determinovanost**) a po konečném počtu kroků algoritmus končí (**konečnost**) s očekávaným výsledkem (**rezultativnost**).

Způsoby zápisu algoritmu

Pro sdílení, úpravu a použití algoritmu je nutná možnost jeho zápisu. Způsobů se v praxi používá několik a volba konkrétního zápisu vždy záleží na charakteru úlohy: (Vaníček, 2007)

- přirozený jazyk (slovní popis, návod, kuchařský recept)
- grafický zápis (vývojový diagram, strukturogram)
- pseudokód (přirozený jazyk doplněný klíčovými slovy)
- programovací jazyk (Taufer, a další, 2001)

2.2 Třídící algoritmy

Při práci s velkým objemem dat je pro další manipulaci výhodné mít tato data setříděna podle námi zvoleného kritéria. Kritérium může být různé, v praxi se nejčastěji setkáváme s tříděním dle abecedy u slovního seznamu nebo dle velikosti u číselného seznamu. Takovéto seřazení nám zásadně ulehčí a zrychlí jednu zcela základní operaci – vyhledávání v seznamu.

Pojem třídění můžeme obecně definovat: *„je dána množina* $S = \{a_1, a_2, \ldots, a_n\}$*, máme najít permutaci P těchto n prvků, která zobrazuje posloupnost* $a_1, a_2, \ldots, a_n$ *do neklesající posloupnosti* $a_{P(1)}, a_{P(2)}, \ldots, a_{P(n)}$*. Číslo i nazveme pozicí prvku* $a_{P(i)}$ *v množině S.“* (Prokop, 2009 str. 37)

Třídící algoritmy jsou tedy algoritmy, jejichž výstupem je setříděná posloupnost vstupních dat, dle předem určeného kritéria.

Řadící x třídící algoritmy

V anglické literatuře se setkáváme s jednotným pojmem *Sorting algorithms*. Rozpory přicházejí po překladu do češtiny, sloveso *sort* lze přeložit hned několika způsoby: uspořádat, třídit, přebírat. Po jazykové stránce není tedy terminologie zcela jednoznačná.

Tento fakt potvrzuje i Virius (Virius, 2002 str. 18) ve své knize: „*Ve skutečnosti jde o řazení prvků podle velikosti, nikoli o rozdělování do tříd. V české programátorské hantýrce se ale zpravidla mluví o třídění, a proto u tohoto termínu zůstaneme. Ostatně anglický termín sorting znamená také třídění, nikoli řazení.“*

Dále můžeme porovnat používání obou výrazů v české odborné literatuře. Například v publikaci *Umění programování* (Knuth, 2008) od vydavatelství Computer Press, se používá výraz řazení. Naopak publikace *Algoritmy v jazyce C a C++* (Prokop, 2009), *Algoritmy, datové struktury a programovací techniky* (Wróblewski, 2004), nebo *Algoritmy v C* (Sedgewick, 2003) potvrzují častější používání pojmu třídění.

2.3 Klasifikace třídících algoritmů

Klasifikaci třídících algoritmů můžeme uchopit dle různých parametrů:

- **dle paměti:**

Tato práce se věnuje pouze **vnitřnímu třídění**, při kterém jsou všechny operace prováděny v operační paměti počítače. Na rozdíl **třídění vnější** má data uložena na záznamovém médiu. To omezuje rychlost práce s těmito daty.

- **dle metody:**

Třídění výměnou je založeno na porovnávání dvou prvků a jejich výměny v případě nesprávného pořadí. Postupným vyměňováním (tříděním) dvojice prvků se seřazuje i celá posloupnost. Způsob volby konkrétních dvou prvků záleží vždy na zvoleném algoritmu. Výměnou třídí Bubblesort, Combsort, Heapsort.

Při **třídění vkládáním** jsou z neseřazeného seznamu postupně prvky vkládány hned na správné místo v setříděné části. Správné zařazení je docíleno postupným porovnáváním zařazovaného prvku s prvky v setříděné části seznamu. Vkládáním třídí Insertsort a Shellsort.

Princip **třídění výběrem** je založený na výběru extrémní (nejmenší či naopak největší) hodnoty. Tato hodnota je následně přesunuta na konec seřazované části. Posupně jsou vyhledány a zařazeny všechny prvky až na poslední, který je zařazen triviálně. Výběrem třídí Selectsort.

Metoda **Třídění rozdělováním** spočívá ve třech základních krocích. Prvním krokem je analýza problému, tedy rozdělení problému na menší podproblémy. Druhý krok je rekurze, která opakuje další rozdělování podproblémů. Rekurzivní opakování probíhá až do okamžiku triviálního rozdělení podproblémů. Po získání triviálních problémů, které již dokážeme řešit, pokračujeme třetím, závěrečným, krokem. Tím je tzv. syntéza, řešením malých problémů se syntetizuje řešení původního problému.

Této metody využívají třídící algoritmy Mergesort a Quicksort. Setřídit celé pole je složitý problém. Proto jej rozdělíme na malé – triviální problémy (pole délky jedna), která jsou řešena - seřazena již triviálně. Z triviálně seřazených dílčích polí postupně sestavujeme seřazené původní pole. (Prokop, 2009)

- **dle stability**

Problematika stability algoritmu se týká řazení seznamů s opakujícími se hodnotami prvků. Zachová-li algoritmus po seřazení prvky stejných hodnot v původním pořadí, označujeme algoritmus za stabilní. Dojde-li při řazení ke změně pořadí prvků stejných hodnot, nazýváme algoritmus nestabilním.

- **dle přirozenosti**

Přirozeným označujeme třídící algoritmus, který dokáže částečně seřazený seznam seřadit rychleji než seznam neseřazený. V opačném případě, u nepřirozených algoritmů, není míra

seřazenosti prvků důležitá. Přirozené algoritmy se v praxi využívají k ověřování seřazenosti seznamů. (Sedgewick, 2003)

	přirozenost	**stabilita**	**metoda**
Bubblesort	ano	ano	výměnou
Insertsort	ano	ano	vkládáním
Selectsort	ne	ne	výběrem
Quicksort	ne	ne	rozdělením
Heapsort	ne	ne	výměnou
Mergesort	ne	ne	rozdělením
Shellsort	ano	ne	vkládáním
Combsort	ano	ne	výměnou

Tabulka 3: Přehled vlastností třídících algoritmů

Upraveno z: http://www.algoritmy.net/article/75/Porovnani-algoritmu

dle složitosti

Hlavním kritériem pro posuzování algoritmů je složitost. Je definována jako počet kroků (elementárních operací) vzhledem k množině vstupních dat. K základní klasifikaci se obvykle využívá tzv. *asymptotická složitost*. Ta je rozdělena do několika tříd složitosti (viz níže) a každý algoritmus dané třídy je vždy pomalejší než algoritmus třídy vyšší bez ohledu na výkonnost počítače.

Tuto skutečnost lze snadno demonstrovat na příkladu. Budeme-li mít dva algoritmy stejného řádu složitosti, první $O(n)$, druhý $O(2n)$. Oba algoritmy jsou lineárního řádu složitosti a stačí pouze druhý algoritmus provést na $2x$ rychlejším stroji a rozdíl je smazán. V případě, kdy algoritmy nejsou stejné složitosti, například první lineárního řádu $O(n)$ a druhý kvadratického řádu $O(n^2)$, tak výkon algoritmu výkonem počítače vyrovnat nedokážeme. Při různých řádech složitosti bude od určitého množství vstupních prvků rychlejší asymptoticky lepší algoritmus bez ohledu na výkonost počítače.

Rozlišujeme tři typy složitostí: **minimální Ω** složitost je nejmenší mez složitosti algoritmu. Této složitosti algoritmus dosahuje při optimálních vstupních datech. Při třídění se může například jednat o již seřazené prvky. **Průměrná Θ** složitost je očekávána funkce pro běžná, náhodná, vstupní data. Zjišťuje se statisticky na základě pravděpodobnosti výskytu odlišných tvarů vstupních dat. Horní mez složitosti algoritmu, tedy nejhorší složitost, které může algoritmus dosáhnout označuje složitost **maximální O**. Při porovnávání různých algoritmů se této funkci používá pro její nezávislost na vstupních datech. (Knuth, 2008)

Řád složitosti je dán dominantní složkou funkce.

- $O(1)$ konstantní
- $O(\log n)$ logaritmická
- $O(n \log n)$ lineárně logaritmická
- $O(n)$ lineární
- $O(n^2)$ kvadratická
- $O(n^c), c > 1$ polynomiální
- $O(c^n)$ exponenciální
- $O(n!)$ faktoriálová

Příklad výpočetní náročnosti

1 operace = 1 µs	**Počet vstupních prvků (n)**							
	10	**20**	**50**	**100**	**1 000**	**1 000 000**	**10^9**	**10^{20}**
$\log n$	1 ns	1 ns	2 ns	2 ns	3 ns	6 ns	9 ns	20 ns
$\sqrt{n}$	3 ns	4 ns	7 ns	10 ns	32 ns	1 µs	32 µs	10 s
n	10 ns	20 ns	50 ns	100 ns	1 µs	1 ms	1 s	3 171 let
n log n	10 ns	26 ns	85 ns	200 ns	3 µs	6 ms	9 s	63 420 let
n^2	100 ns	400 ns	3 µs	10 µs	1 ms	17 min	32 let	
n^3	1 µs	8 µs	125 µs	1 ms	1 s	32 let		
2^n	1 µs	1 ms	13 dní					
3^n	59 µs	3 s	22 mil.let					
$n!$	4 ms	77 let						
n^n	10 s							

Tabulka 4: Příklad výpočetní náročnosti

Z tabulky vidíme, že náročnost algoritmu určuje nejen dobu, kterou budeme na výsledek čekat, ale především, zda se výsledku lze dočkat a algoritmus prakticky využít. Pro informativní přehled jsem použil výpočetní rychlost jedné operace $1\mu s$. Za prakticky použitelné se obvykle považují algoritmy třídy složitosti nejvýše polynomiální.

3 Realizace třídících algoritmů

Jednotlivé třídící algoritmy patří k nejznámějším standartním algoritmům. Věnuje se jim mnoho publikací od učebnic pro začátečníky až po odbornou, především cizojazyčnou, literaturu. Pro vypracování uceleného přehledu základních třídících algoritmů byla použita tato literatura: (Sedgewick, 2003), (Prokop, 2009), (Wróblewski, 2004), (Töpfer, 1995), (Čápka).

3.1 Bubblesort

Princip je založený na porovnávání dvou sousedních prvků číselného pole. Pole procházíme z leva doprava a postupně porovnáváme všechny sousední prvky. Nejsou-li ve správném pořadí, tak dvojici prvků vyměníme. Po ukončení iterace se vždy na konec pole dostane nejmenší prvek.

Ve schématu vidíme průběh řazení krok po kroku. Oranžovou barvou jsou vyznačeny prvky, které se právě porovnávají. Zelený prvek je již správně zařazený.

1	3	2	4	5
3	1	2	4	5
3	2	1	4	5
3	2	4	1	5
3	2	4	5	1
3	2	4	5	1
3	4	2	5	1
3	4	5	2	1
4	3	5	2	1
4	5	3	2	1
5	4	3	2	1

Obrázek 7: Schéma průběhu Bubblesortu

Pole se nám dělí na dvě poloviny – seřazenou a neseřazenou část. Délka neseřazené části se po každé iteraci o jeden prvek zmenší, až nakonec bude obsahovat jen dva prvky. Seřazením poslední dvojice prvků je řazení ukončeno.

Rozbor

Celkový počet iterací je $(n-1)$. Pro každé seřazované číslo je potřeba jedna iterace s výjimkou posledního, které je zařazeno triviálně. Počet porovnání (vnitřní cyklus) v každé iteraci se provede $(n-1)+(n-2)+\cdots+(1) krát$.

Ve schématu názorně vidíme, že proběhnou 4 tedy $(n-1)$ iterace (vnější cykly) a v nich postupně 4, 3, 2 a 1 porovnání (vnitřní cykly).

Podle vzorce součtu prvků aritmetické posloupnosti $s=\frac{n}{2}(a_1+a_n)$ a po dosazení platí $s=\frac{n-1}{2}(n-1+1)=\frac{n^2-n}{2}$. Protože lineární funkce (n) i konstanta (2) rostou asymptoticky pomaleji než kvadratická funkce (n^2) můžeme je zanedbat.

Počet porovnání bude vždy stejný, jedná se tedy o nejlepší i zároveň nejhorší možný výsledek a **výsledná asymptotická složitost je $\boldsymbol{\theta(n^2)}$.**

Kód

Kód se skládá ze dvou vnořených cyklů se známým počtem opakování. Vnitřní cyklus porovnává a případně vyměňuje sousední prvky pole. Počet iterací algoritmu řídí vnější cyklus. Podmínka ($pole.length - i - 1$) zajišťuje zkrácení porovnávané části o část již seřazenou.

```
//--------------BubbleSort--------------
public static void BubbleSort(int[] pole){
  for (int i = 0; i < pole.length - 1; i++) {
    for (int j = 0; j < pole.length - i - 1; j++) {
      if(pole[j] < pole[j+1]){
        int pom = pole[j];
        pole[j] = pole[j+1];
        pole[j+1] = pom;
      }
    }
  }
}
```

3.2 Selectsort

Selectsort (Selection sort) vychází z principu, že v neseřazené části pole vyhledáme největší prvek a zařadíme jej na konec části seřazené.

Konkrétně první prvek neseřazeného zvolíme za největší a porovnáváme s ostatními prvky, najdeme-li prvek větší, zvolíme jej za největší a pokračujeme v porovnávání. Po porovnání celého pole s největším prvkem si můžeme být jisti, že jsme našli největší prvek neseřazeného pole. Proto jej přesuneme na konec seřazené části, respektive jej vyměníme s prvním prvkem části neseřazené.

Ve schématu vidíme postup a jednotlivé kroky, červeně jsme označili největší prvek, oranžově prvek porovnávaný a zelený je již prvek seřazený.

1	3	2	4	5
1	3	2	4	5
1	3	2	4	5
1	3	2	4	5
1	3	2	4	5
5	3	2	4	1
5	3	2	4	1
5	3	2	4	1
5	4	2	3	1
5	4	2	3	1
5	4	3	2	1
5	4	3	2	1

Obrázek 8: Schéma průběhu Selectsortu

Rozbor

Celkový počet iterací je $(n-1)$. Pro každé seřazované číslo je potřeba jedna iterace s výjimkou posledního, které je zařazeno triviálně. Počet porovnání (vnitřní cyklus) v každé iteraci se provede $(n-1)+(n-2)+\cdots+(1) krát$.

Asymptotická složitost Selectsortu je shodná se složitostí Bubblesortu tedy **výsledná asymptotická složitost je $\boldsymbol{\theta(n^2)}$.**

Přesto je Selectsort v obecném případě rychlejší než Bubblesort. Důvodem je počet výměn prvků. Zatímco Bubblesort vyměňuje každé dva špatně seřazené sousední prvky a tím v jedné iteraci může provést hned několik výměn, Selectsort výměnu v každé iteraci provede maximálně jedenkrát.

Kód

Kód se skládá ze dvou vnořených cyklů se známým počtem opakování. Vnitřní for cyklus porovnává největší prvek (maxIndex) s dalšími prvky pole. Podmíněný příkaz If nám v případě nalezení většího prvku, tento prvek označí jako největší.

Vnější for cyklus nám zajišťuje opakování iterací, zvolení vždy nového (prvního) největšího prvku a na závěr prohození největšího nalezeného prvku s prvním neseřazeným prvkem.

```
//--------------Select sort--------------
public static void selectionSort(int[] pole) {
  for (int i = 0; i < pole.length - 1; i++) {
    int maxIndex = i;          //index nejvetsiho prvku
    for (int j = i + 1; j < pole.length; j++) {
      if (pole[j] > pole[maxIndex]) maxIndex = j;
    }
    int pom = pole[i];
    pole[i] = pole[maxIndex];
    pole[maxIndex] = pom;
  }
}
```

3.3 Insertsort

Princip Insertsortu (Insertion sortu) je v postupném plnění seřazené části. Zvolíme první prvek, který je triviálně zařazen v seřazené části pole. Následující prvek porovnáváme se seřazenou částí pole a zařadíme jej na správné místo za první větší prvek. Tento postup opakuje až do zařazení posledního prvku.

Ve schématu vidíme první prvek, který již považujeme za seřazený (zelená). Postupným porovnáváním prvkům (oranžová) prvek zařadíme na správné místo v seřazené části.

1	3	2	4	5
1	3	2	4	5
3	1	2	4	5
3	1	2	4	5
3	2	1	4	5
3	2	1	4	5
3	2	1	4	5
3	2	4	1	5
3	4	2	1	5
4	3	2	1	5
4	3	2	1	5
4	3	2	5	1
4	3	5	2	1
4	5	3	2	1
5	4	3	2	1

Obrázek 9: Schéma průběhu Insertsortu

Rozbor

Celkový počet iterací je $(n-1)$. Pro každé seřazované číslo je potřeba jedna iterace s výjimkou posledního, které je zařazeno triviálně. Počet porovnání (vnitřní cyklus) je závislý na uspořádání prvků a nedá se pevně stanovit. V nejlepším případě, kdy je pole seřazené, algoritmus v každé iteraci provede jediné porovnání se sousedním prvkem. **Minimální asymptotická složitost je $\Omega(n)$.**

Naopak v nejhorším případě může v iteraci nastat postupně opakování až $(n-1)+(n-2)+\cdots+(1) krát$. V tomto případě dosahuje algoritmus **maximální asymptotickou složitost $O(n^2)$.**

Nejvíce se Insertsort hodí pro třídění již částečně seřazených prvků nebo ke kontrole, zda jsou prvky seřazeny správně.

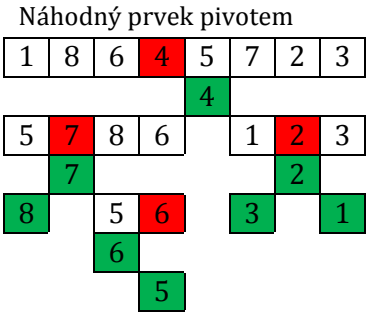
Kód

Kód je tvořen dvěma vnořenými cykly. Vnitřní cyklus *while* má zdvojenou podmínku, která zabezpečuje porovnání zařazovaného prvku (x) s prvky předchozími a zároveň hlídá hranici pole, abychom nepřekročili jeho rozsah. Po nalezení většího prvku se náš porovnávaný prvek (x) zařazuje a vnější cyklus zahajuje další iteraci.

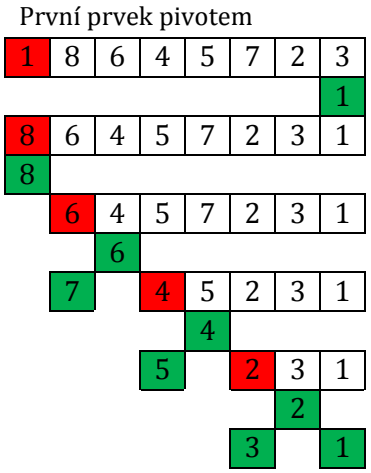
```
// ----------------- Insert sort ---------------- //


      for (int i = 0; i < pole.length - 1; i++) {
        int j = i + 1;
        int x = pole[j];
        while (j > 0 && x > pole[j-1]) {
          pole[j] = pole[j-1];
          j--;
        }
        pole[j] = x;
      }
```

3.4 Quicksort

Princip je založený na volbě referenčního prvku tzv. pivota. Pivota můžeme volit zcela libovolně, může to být první prvek, poslední či třeba náhodně zvolený. Nyní pole uspořádáme tak, aby všechny prvky menší než pivot byly na jedné straně pole a všechny větší prvky na straně druhé. Tak nám vznikne část pole menších prvků, část pole větších prvků a samotný pivot, který je již správně zařazený. Stejným způsobem pokračuje i v obou vzniklých částech, opět zvolíme nové pivoty a rozdělíme část pole. Takto pokračuje až do triviálně řešitelných problémů.



Obrázek 10: Schéma průběhu Quicksortu 1



Obrázek 11: Schéma průběhu Quicksortu 2

Volba pivota

Obě schémata průběhu, lišící se zvoleným pivotem, ukazují, jak je volba pivota důležitým parametrem pro délku seřazování. Zda zvolíme, první či poslední náhodný prvek důležité není, ale záleží na velikosti zvoleného prvku. Pro co nejrychlejší uspořádání je potřeba, aby se pole pravidelně půlilo, tedy aby počet menších i větších prvků než pivot byl přibližně stejný. Z toho plyne, že ideálně zvolený pivot je vždy medián seřazovaných hodnot.

Rozbor

Jak je výše naznačeno, složitost algoritmu závisí na volbě pivota. Při ideální volbě a dělení prvků do binárního stromu potřebujeme pouze $(\log_2 n)$ iterací. Při každém volání se vyměňuje až n prvků, tedy výsledná složitost je $\boldsymbol{\Omega(n \log_2 n)}$.

Pokud naopak budeme pivoty volit nevhodně, dostaneme se až na n iterací, ve kterých provedeme až n výměn. Tedy nejhorší asymptotická náročnost je $\boldsymbol{O(n^2)}$.

Quicksort je vhodný pro řazení rozsáhlých polí, naopak pro malá pole je naprosto nevhodný. Je to opět z důvodu volby pivota. V rozsáhlém poli je větší pravděpodobnost dostatečného počtu větších i menších prvků než pivot.

Kód

V kódu vidíme volbu pivota (piv) jako první prvek pole. Pro výměnu prvků používáme metodu *swap*, která nám vždy zaručí správné zařazení větších prvků před pivota a menších prvků za něj.

Pozornost si zaslouží rekurzivní volání hlavní metody Quicksort s rozdílnými parametry. To je z důvodu volání metody zvlášť pro levou část pole a zvlášť pro část pravou.

```
  public static void quicksort(int[] pole , int levy , int pravy)
  {
    if(levy < pravy)
    {
      int piv = levy;
      for(int i = levy + 1; i < pravy; i++){
        if(pole[i] > pole[levy])
          swap(pole, i, ++piv);

      }
      swap(pole,  levy, piv);
      quicksort(pole, levy, piv);
      quicksort(pole, piv + 1, pravy);
    }
  }

  private static void swap(int[] pole, int levy, int pravy){
    int a = pole[pravy];
    pole[pravy] = pole[levy];
    pole[levy] = a;
  }
}
```

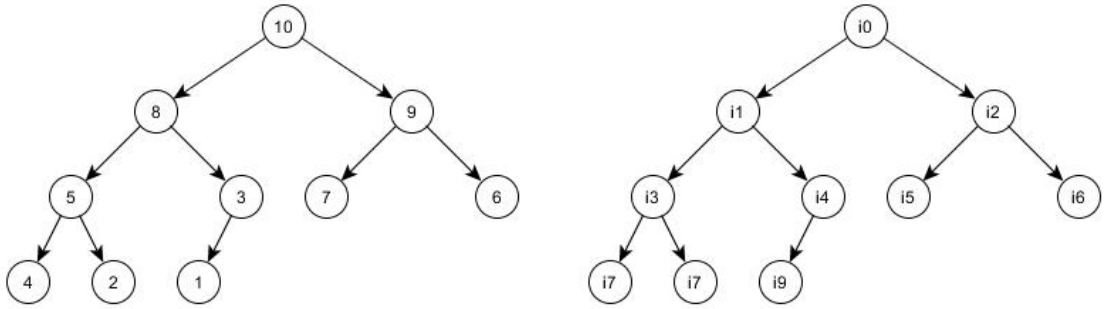
3.5 Heapsort

Heapsort je založený na podobném principu jako Selectsort, tedy vyhledá největší prvek z pole a přesouvá jej na konec seřazené části. Slabou stránkou Selectsortu je právě ono vyhledávání největšího prvku. Pro každý prvek docházelo k procházení celého pole a k porovnání každého prvku. To řeší právě Heapsort, pomocí haldy reprezentuje prvky stejně jako pole, ale největší prvek dokáže nalézt v konstantním čase.

Halda je triviální binární strom s následujícími vlastnostmi:

- Všechny vrstvy haldy jsou plně obsazeny prvky. Poslední vrstva nemusí být plná.
- Poslední vrstva je plněna z leva.
- Oba přímí následníci jsou vždy menší nebo rovny přímému předchůdci.

Z poslední vlastnosti jasně vyplývá, že ve vrcholu (kořenu) haldy bude vždy uložen největší prvek. Navíc je halda vlastnost rekurzivní, tedy všechny podstromy jsou také haldy. Tyto vlastnosti nám umožňují haldu využít pro řazení prvků. (Töpfer, 1995)



index	i0	i1	i2	i3	i4	i5	i6	i7	i8	i9
hodnota	10	8	9	5	3	7	6	4	2	1

Obrázek 12: Halda

Tvorba haldy

Haldu lze díky jejímu vyvážení sestrojit přímo v poli. Prvky pole jsou v haldě řazeny od shora dolů a zleva doprava. Kořen haldy má tedy index 0 a poslední prvek haldy je zároveň i posledním prvkem pole. Navíc platí i vztahy mezi indexy přímých následníků a předchůdců. Je-li i index předchůdce, pak index levého přímého následovníka je $2i + 1$ a pravého přímého následovníka $2i + 2$. Je-li i index jednoho z následovníků, pak index přímého předchůdce je $(i - 1)/2$ (dělení celočíselné).

Oprava haldy

Máme sestavenou „haldu“, která ale nesplňuje její podmínky. Pro opravení haldy použijeme funkci *repair up*, kterou budeme postupně volat pro prvky od kořene dolů. Funkce ověří, zda je splněna podmínka haldy, tedy zda předchůdce je větší než přímý následovník. V případě, kdy podmínka splněna není, prvky prohodí a funkce se opakuje,

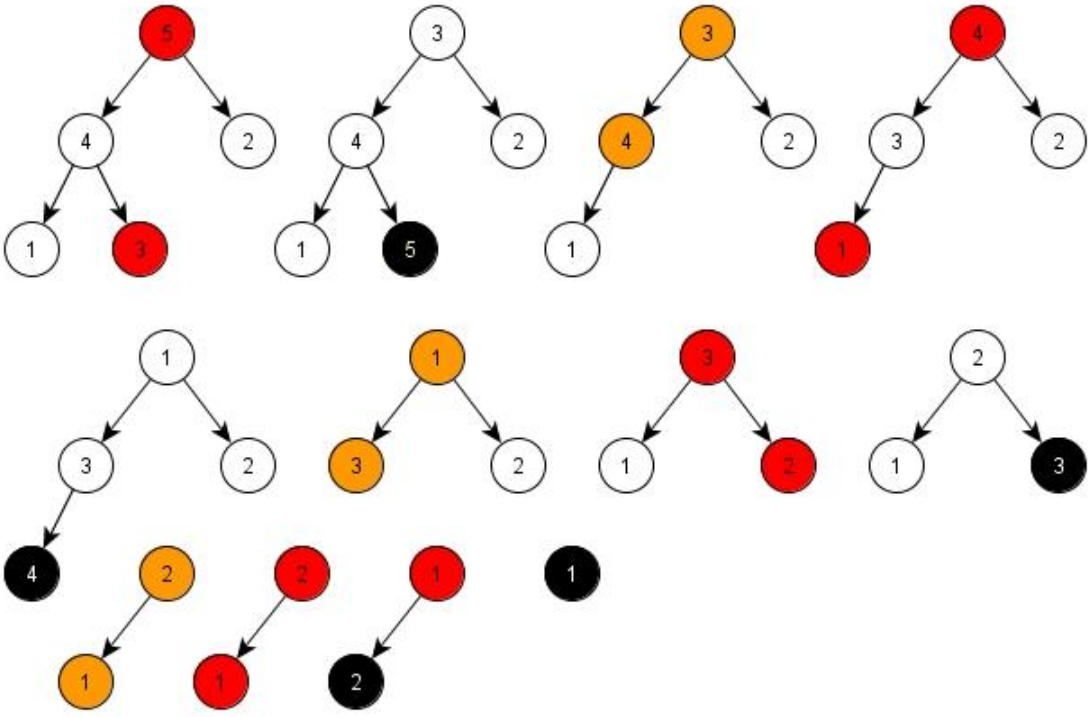
aby se předcházelo přenášení problému o úroveň výše. Po zkontrolování celého stromu, získáváme haldu.

Třídění

V haldě vezmeme největší prvek (kořen) a vyměníme jej s prvkem na posledním místě haldy. Přesunutý největší prvek (původní kořen) již zanedbáme, je to seřazený prvek v seřazené části pole.

Tím jsme si ale jistě rozbili haldu a je potřeba ji opět opravit, aby respektovala podmínky haldy. Opravu provedeme velice podobnou funkcí *repair down.* Tuto funkci zavoláme na kořen haldy a v případě nálezu většího následovníka vyměníme předchůdce s větším ze dvojice přímých následovníků. Z důvodu možnosti posunutí problému o úroveň níže ověřování opakujeme. Po skončení opravy získáváme opravenou haldu a můžeme pokračovat další iterací.

Postup tvorby pole i následné třídění společně s opravami haldy presentuje následující schéma:



Obrázek 13: Heapsort - průběh funkce down

Rozbor

Odtržení maxima i jeho prohození s posledním prvkem jsou procedury, které algoritmus časově nezatěžují. Časově náročnou operací je opětovné uspořádání haldy (funkce *repair down*), která se bude provádět po každém odebrání kořene, tedy n krát. V každém svém zavolání se bude provádět maximálně $\log_2 n$ krát. To je způsobeno strukturou haldy, která tvoří binární strom. Hloubka stromu je rovna $\log n$ a protože je binární, tedy předchůdce má vždy dva přímé následovníky, jedná se o logaritmus o základu dva.

Časová náročnost funkce repair down je totožná. V každé iteraci je jednou volána funkce *repair up* a *n* krát funkce *repair down*. **Výsledná asymptotická náročnost je tedy** $\mathbf{O}(\boldsymbol{n}\ \mathbf{log}_2\ \boldsymbol{n})$.

Kód

Kód se skládá z jednotlivých metod. Nejprve je volána metoda Heapsort, která hned volá metodu *heapify*. Tato metoda vytvoří haldu a následuje volání metod *down* a *up*, které haldu postupně opravují po odebírání kořenů – největších prvků pole.

```
  //--------------Heap sort--------------
//oprava haldy nahoru
  public static void up(int[] list, int i) {
    int child = i; //ulozim syna
    int parrent, temp;
    while (child != 0) {
      parrent = (child - 1) / 2; //otec
      if (list[parrent] < list[child]) { //detekce chyby
        temp = list[parrent]; //prohozeni syna s otcem
        list[parrent] = list[child];
        list[child] = temp;
        child = parrent; //novy syn
      }
      else
        return;
    }
  }

//oprava haldy dolu
  public static void down(int[] list, int last) {
    int parrent = 0;
    int child, temp;
    while (parrent * 2 + 1 <= last) {
      child = parrent * 2 + 1;
      // pokud je vybran mensi syn
      if ((child < last) && (list[child] < list[child + 1]))
        child++; //vybereme toho vetsiho
      if (list[parrent] < list[child]) { //detekce chyby
        temp = list[parrent]; //prohozeni syna s otcem
        list[parrent] = list[child];
        list[child] = temp;
        parrent = child; //novy otec
      }
      else
        return;
    }
  }
```

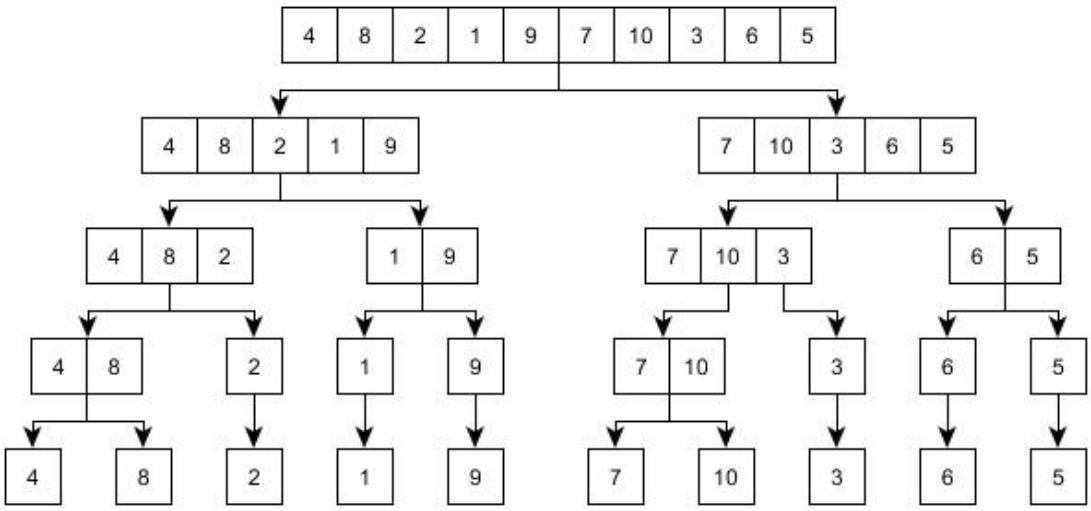
```
// postaveni haldy z pole
  public static void heapify(int[] list) {
    for (int i = 1; i < list.length; i++)
      up(list, i);
  }

// samotne tridění
  public static void heapsort(int[] list) {
    heapify(list);
    int index = list.length - 1; // posledni prvek
    int temp;
    while (index > 0) {
      temp = list[0]; // prohozeni posledního prvku s maximem
      list[0] = list[index];
      list[index] = temp;
      index -= 1; //nastaveni noveho posledniho prvku
      down(list, index);
```

3.6 Mergesort

Základní princip Mergesortu je podobný jako princip Quicksortu. Pole s prvky je rozděleno na dvě stejná pole. Toto dělení pole je rekurzivně opakováno, dokud nám nevzniknou pouze jednoprvková pole, která jsou triviálně seřazena. Tato jednoprvková pole postupně spojíme-slejeme (angl. merge) na seřazená dvouprvková pole, dále na čtyřprvková a tak dále, až získáme dvě velká seřazená pole, které po spojení vrátí seřazené původní pole.

Dělení

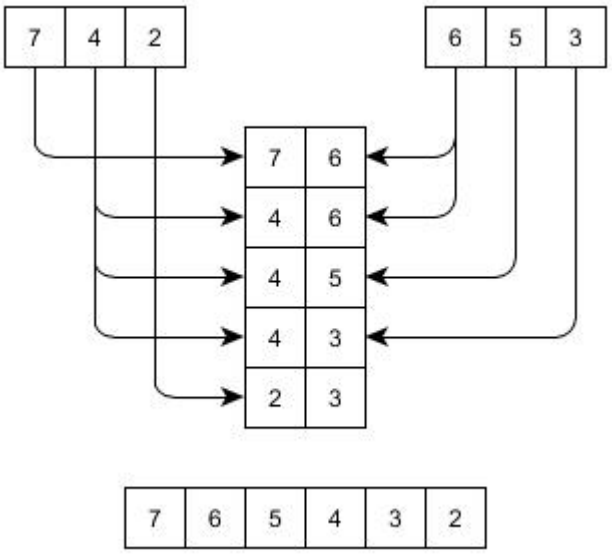
Průběh dělení pole, které je prováděno rekurzivně až na pole jednotkové velikosti, ukazuje následující schéma.



Obrázek 14: Schéma průběhu dělení Mergesortu

Slévání

Princip slévání dvou polí je jednoduchý. Máme dvě pole A a B, která jsou seřazena. Poté porovnáme první prvky obou polí a větší přesuneme na první pozici pomocného pole, z původního pole jej vymažeme. Opět porovnáme první prvky obou polí a pokračujeme obdobně. Až jedno z polí bude prázdné, přesuneme zbytek druhého pole na konec pole pomocného. Průběh jednoho slévání dvou tříprvkových polí ukazuje následující schéma:



Obrázek 15: Schéma průběhu slévání Mergesortu

Rozbor

Protože mergesort při dělení pole vytváří strukturu odpovídající binárnímu stromu, budeme z tohoto faktu vycházet při zkoumání náročnosti algoritmu. Můžeme tedy analyzovat potřebný čas na zpracování n prvků v jednotlivých hloubkách binárního stromu.

V každém uzlu binárního stromu pracujeme s polem délky $\frac{n}{2^i}$, proto potřebný čas pro zpracování uzlu je $O(\frac{n}{2^i})$, kde n je počet prvků a i je hloubka stromu. V každé holubce i má navíc strom právě 2^i uzlů. A z toho nám vyplývá, že strávený čas v každé hloubce stromu je právě $O(2i\frac{n}{2^i})$, to můžeme asymptoticky zkrátit na $O(n)$. Počet hloubek odpovídá výšce binárního stromu, tedy $\log_2 n$.

Celková nejhorší **asymptotická časová náročnost je $O(n \log_2 n)$.**

Kód

Samotný kód je na první pohled nepřehledný, ale jsou pouze rekurzivně volané operace, které jsme si výše popsali.

Metoda mergeSort dělí pole a následně se rekurzivně volá na nově vzniklá pole left a right. Poté, co jsou obě pole rozdělena na jednoprvková, je volána metoda merge, která tyto pole slévá a postupně porovnává.

```
  //--------------MergeSort--------------
// sliti dvou setridenych poli do jednoho
  public static void merge(int[] pole, int[] left, int[] right) {
    int i = 0;
    int j = 0;
    // dokud nevyjedeme z jednoho z poli
    while ((i < left.length) && (j < right.length)) {
      // dosazeni toho mensiho prvku z obou poli a posunuti inde
      if (left[i] > right[j]) {
        pole[i + j] = left[i];
        i++;
      }
      else {
        pole[i + j] = right[j];
        j++;
      }
    }
    // doliti zbytku z nevyprazdneneho pole
    if (i < left.length) {
      while (i < left.length) {
        pole[i + j] = left[i];
        i++;
      }
    }
    else {
      while (j < right.length) {
        pole[i + j] = right[j];
        j++;
      }
    }
  }

// samotne trideni
  public static void mergeSort(int[] pole) {
    if (pole.length <= 1) //podminka rekurze
      return ;
    int center = pole.length / 2; //stred pole
    int[] left = new int[center]; //vytvoreni a naplneni leveho pole
    for (int i = 0; i < center; i++)
      left[i] = pole[i];
    int[] right = new int[pole.length - center]; //vytvoreni a naplneni praveho pole
    for (int i = center; i < pole.length; i++)  //vytvoreni a naplneni praveho pole
      right[i - center] = pole[i];
    mergeSort(left); // rekurzivni zavolani na obe nova pole
    mergeSort(right);
    merge(pole, left, right); //sliti poli
  }
```

3.7 Shellsort

Algoritmus, který vymyslel roku 1959 Donald Shell, modifikuje Insertsort, který si uchovává seřazenou část pole a v každém kroku zařadí přímo sousedící prvek s touto částí pole na správné místo.

Základem Shellsortu je využívání tzv. snižujícího se přírůstku. Tedy algoritmus neřadí sousední prvky, ale dvojici prvků, mezi kterými je určitá mezera zmenšující se po každé iteraci. Až v okamžiku, kdy se mezera sníží na velikost jedna, dochází k porovnávání sousedních prvků a algoritmus degeneruje na Insertsort.

Zásadní výhodou Shellsortu je eliminování problému nazvaný *Zajíc a želva*. Prvky vysokých a nízkých hodnot jsou velmi rychle přesunuty do odpovídající části pole. Tím se snižuje počet přesunovaných prvků v poslední iteraci, kdy je mezera rovna jedné a extrémně velké (resp. malé) prvky potřebují vysoký počet výměn k přesunutí na správné místo v poli.

1	8	10	4	5	7	2	9	6	3	mezera
1	8	10	4	5	7	2	9	6	3	8
6	8	10	4	5	7	2	9	1	3	
6	8	10	4	5	7	2	9	1	3	4
6	8	10	4	5	7	2	9	1	3	
6	8	10	4	5	7	2	9	1	3	
6	8	10	4	5	7	2	9	1	3	
6	8	10	9	5	7	2	4	1	3	
6	8	10	9	5	7	2	4	1	3	
6	8	10	9	5	7	2	4	1	3	1
8	6	10	9	5	7	2	4	1	3	
10	8	6	9	5	7	2	4	1	3	
10	9	8	6	5	7	2	4	1	3	
10	9	8	6	5	7	2	4	1	3	
10	9	8	7	6	5	2	4	1	3	
10	9	8	7	6	5	2	4	1	3	
10	9	8	7	6	5	4	2	1	3	
10	9	8	7	6	5	4	2	1	3	
10	9	8	7	6	5	4	3	2	1	
10	9	8	7	6	5	4	3	2	1	

Obrázek 16: Schéma průběhu Shellsortu

Schéma znázorňuje výhodnost Shellsortu oproti Insertsortu a eliminaci problému *Zajíc a Želva.* To výrazně omezuje počet výměn a celkově urychluje celý algoritmus. Samotné řazení již probíhá shodně s Insertsortem, zařazovaný prvek se porovnává s předchozími do doby, než se se najde větší a poté se zařadí na pozici před něj. Oranžově zvýrazněné prvky jsou právě porovnávány a červené nevyhovují podmínce a jsou vyměněny. Zelený prvek je již zařazen na místě správném. Z důvodu názornosti na malém množství prvků byla velikost první mezery zvolena osm.

Volba velikosti mezery

Zakladatel algoritmu Donald Shell navrhl mezeru o velikosti $n/2$, která by se každou další iterací nadále půlila. Tímto postupem se ovšem budou prvky na sudých místech pole s prvky na lichých místech porovnávat až v poslední iteraci s velikostí mezery jedna.

Proto se další matematici a programátoři snažili najít vhodnější způsob volby velikosti mezery. Jedním z nich byl Hall Hibbard, který zkoumal efektivitu algoritmu s posloupností velikosti mezery $2^k - 1$, kde k je pořadí iterace. S touto volbou mezery dosáhl složitosti $O(n^{\frac{3}{2}})$. O něco lepšího výsledku dosáhl Sedgewick s posloupností $9 \cdot 4^i - 9 \cdot 2^i$, která dosahovala složitosti $O(n^{4/3})$. Dále se experimentovalo s Fibonacciho posloupností i s vlastnostmi zlatého řezu.

Nejstabilnější výsledky presentoval Marcin Ciura s posloupností (1; 4; 10; 23; 57; 132; 301; 701; 1750; …; $mezera \cdot 2{,}2$). Empiricky zjištěná **celková asymptotická náročnost je $O(n^{4/3})$.** (Prokop, 2009)

Kód

Základem kódu je samozřejmě Insertsort, který je doplněný o zmenšující se mezeru. Hned v prvním řádku se velikost mezery definuje jako polovina délky pole. Následně jsou porovnávány prvky postupně pomocí vnořeného cyklu porovnávány prvky. V samotném závěru je velikost mezery opět změněna koeficientem.

```
// ----------------- ShellSort ----------------
    int mezera = pole.length / 2; //deleni mezery 2
    while (mezera > 0) {
      for (int i = 0; i < pole.length - mezera; i++) { // insertion sort
        int j = i + mezera;
        int tmp = pole[j];
        while (j >= mezera && tmp > pole[j - mezera]) {
          pole[j] = pole[j - mezera];
          j -= mezera;
        }
        pole[j] = tmp;
      }
      if (mezera == 2)  //zmena velikosti mezery
        mezera = 1;
      else
        mezera /= 2.2; //zmena velikosti mezery
    }
```

3.8 Combsort

Algoritmus vymyslel v roce 1980 polský inženýr Włodzimierz Dobosiewicz. Vycházel z Bubblesortu a přestože má s Bubblesortem stejnou kvadratickou složitost, díky eliminaci problému *Zajíc a želva* je v praxi rychlejší.

Princip je velice podobný Shellsortu, který je inovací Insertsortu. I zde se největší slabina Bubblesortu, tedy porovnávaní vždy sousedních prvků, řeší zavedením zmenšující se mezery. Tato mezera eliminuje problém *Zajíc a želva* tím, že prvky vysokých a nízkých hodnot se rychle přesunou do správné části pole. V poslední iteraci, kdy je mezera zmenšena na velikost jedna a algoritmus degeneruje na Bubblesort, se již přesouvá pouze malé množství prvků a to vždy maximálně o jednu pozici.

1	8	6	4	5	7	2	3	mezera
1	8	6	4	5	7	2	3	6
2	8	6	4	5	7	1	3	
2	8	6	4	5	7	1	3	4
5	8	6	4	2	7	1	3	
5	8	6	4	2	7	1	3	
5	8	6	4	2	7	1	3	
5	8	6	4	2	7	1	3	3
5	8	6	4	2	7	1	3	
5	8	6	4	2	7	1	3	
5	8	7	4	2	6	1	3	
5	8	7	4	2	6	1	3	
5	8	7	4	3	6	1	2	2
7	8	5	4	3	6	1	2	
7	8	5	4	3	6	1	2	
7	8	5	4	3	6	1	2	
7	8	5	6	3	4	1	2	
7	8	5	6	3	4	1	2	
7	8	5	6	3	4	1	2	1
8	7	5	6	3	4	1	2	
8	7	5	6	3	4	1	2	
8	7	6	5	3	4	1	2	
8	7	6	5	3	4	1	2	
8	7	6	5	4	3	1	2	
8	7	6	5	4	3	1	2	
8	7	6	5	4	3	2	1	

Obrázek 17: Schéma průběhu Combsortu

Volba velikosti mezery

Celková časová náročnost opět závisí na správné volbě mezery. Experimentovalo se s mnoha různými posloupnostmi velikostí, ale za nejvýhodnější velikost je označována mezera, jejíž velikost je postupně dělena koeficientem 4/3.

Schéma nám ukazuje postup porovnávání prvků (oranžová) a jejich případnou výměnu (červená). Pro seřazení osmi prvků bylo za celý průběh zapotřebí 10 výměn. To dokazuje

výhodu oproti Bubblesortu, který by potřeboval jen pro správné zařazení prvního prvku velikosti jedna výměn sedm.

Rozbor

Rozbor algoritmu Combsort je pochopitelně podobný Bubblesortu. Vnitřní cyklus se vždy provede právě $(n - mezera)$ krát. Počet opakování vnějšího cyklu (*while*), ve kterém se také provádí změna velikosti mezery, lze vyjádřit jako počet prvků posloupnosti s koeficientem $q = 4/3$, konkrétně $(n; n \cdot q; n \cdot q^2; \dots; 1)$. Tedy počet iterací lze vyjádřit vztahem $\frac{\log n}{\log 4/3} - 1$. Po asymptotickém zjednodušení uvedených vztahů docházíme k **celkové časové složitosti O($n \log n$).**

Poznámka: U Combsort lze často nalézt uvedenou náročnost O(n^2). Přestože tento údaj není asymptoticky chybný, je nepřesný.

Kód

V kódu na první pohled rozeznáme vnitřní for cyklus, který je běžným Bubblesortem. Vnější cyklus while, který nám opakuje iterace, v tomto algoritmu navíc mění velikost mezery. Po ukončení iterace s velikostí mezery jedna algoritmus končí a pole je seřazené.

```
//--------------CombSort--------------

public static void combSort(int[] pole, int count, int cc) {
  int mezera = pole.length;
  while (mezera != 1) {
    mezera /= 1.33; //zmenseni mezery o 4/3

    for (int i = 0; i + mezera < pole.length; i++) {
      if (pole[i] < pole[i + mezera]) {
        int tmp = pole[i];
        pole[i] = pole[i + mezera];
        pole[i + mezera] = tmp;
      }
    }
  }
}
```

4 Porovnání jednotlivých algoritmů

V této praktické části bakalářské práce budu prezentovat výsledky porovnávání jednotlivých třídících algoritmů. Při testování budu sledovat 3 základní parametry - počet porovnání prvků, počet výměn prvků a dobu seřazování v milisekundách. Po srovnání jednotlivých výsledků testů snadno získáme přehled o vhodnosti využití daných algoritmů pro konkrétní případy. Kódy všech testovaných programů jsou přílohou této práce.

Zmíněné parametry budeme sledovat na 5 vzorových polích:

- 10 náhodných prvků
- 1 000 náhodných prvků
- 100 000 náhodných prvků
- 1 000 seřazených prvků
- 1 000 antiseřazených prvků – seřazené v opačném pořadí

1. Srovnání dle vlastností algoritmu

	Časová náročnost				
	min Ω	**avg θ**	**max O**	**přirozenost**	**stabilita**
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$	ano	ano
Insertsort	$O(n)$	$O(n^2)$	$O(n^2)$	ano	ano
Selectsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	ne	ne
Quicksort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	ne	ne
Heapsort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	ne	ne
Mergesort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	ano	ano
Shellsort			$O(n^{4/3})$	ano	ne
Combsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	ano	ne

Tabulka 5: Vlastnosti třídících algoritmů

Upraveno z: http://www.algoritmy.net/article/75/Porovnani-algoritmu

Jsou-li v literatuře srovnávány jednotlivé algoritmy, vždy jsou porovnávány dle třídy časové složitosti. V přehledové tabulce máme připomenuty všechny vlastnosti jednotlivých algoritmů. Takovýto přehled může být klíčem pro zvolení nejvhodnějšího algoritmu pro řazení konkrétních dat.

Při velkém objemu vstupních dat vždy budeme volit algoritmus s nejmenším řádem časové složitosti, v případě částečně setříděných vstupních dat volíme algoritmus přirozený a při vstupních datech s častým případem shodných hodnot využijeme některý ze stabilních algoritmů. Při větším počtu prvků ($n > 100\,000$) je vliv správně zvoleného algoritmus po časové stránce velmi výrazný.

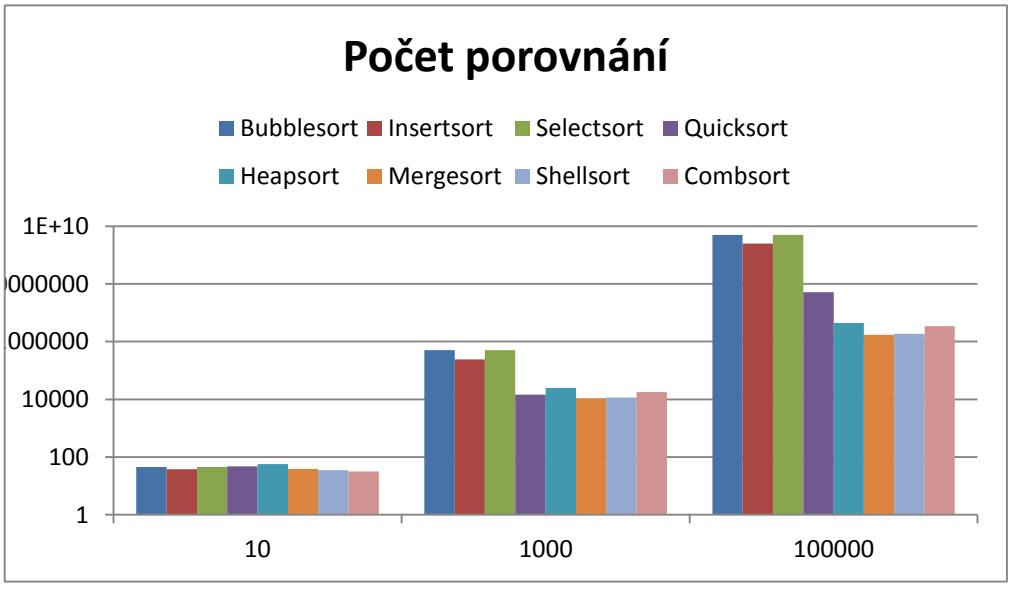
2. Srovnání podle počtu porovnání a výměn

Dle velikosti pole

	$n = 10$		$n = 1\,000$		$n = 100\,000$	
	porovnání	**výměn**	**porovnání**	**výměn**	**porovnání**	**výměn**
Bubblesort	45	29	499 500	239 167	4 999 950 000	2 473 738 990
Insertsort	38	29	240 166	239 167	2 473 838 989	2 473 738 990
Selectsort	45	9	499 500	999	4 999 950 000	99 999
Quicksort	48	23	14 541	4 447	51 048 795	457 481
Heapsort	57	30	24 569	9 430	4 444 938	1 612 080
Mergesort	39	34	10 685	9 976	1 733 049	1 668 928
Shellsort	35	13	11 689	4 599	1 859 535	551 189
Combsort	32	13	18 010	3 064	3 397 054	320 274

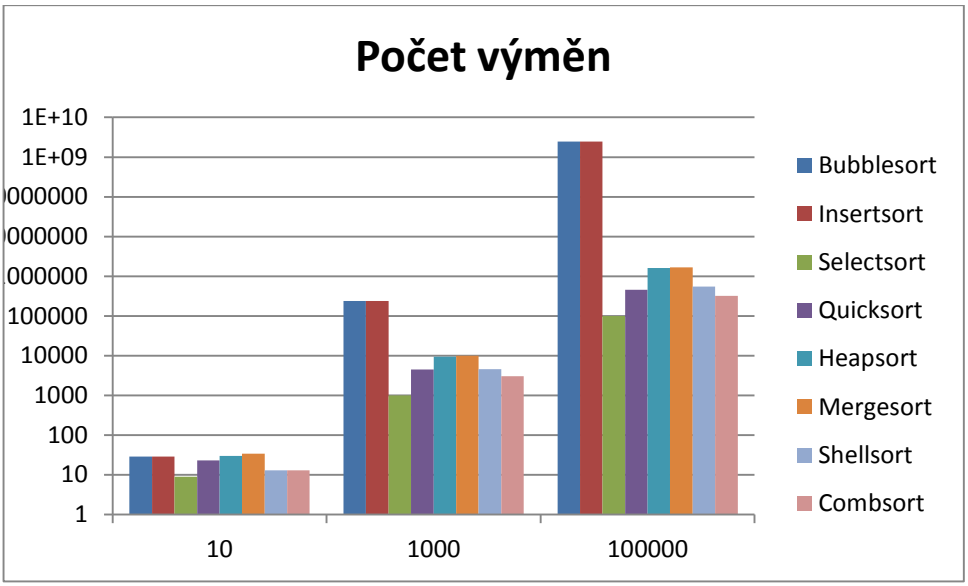
Tabulka 6: Počet porovnání a výměn dle velikosti

Při prvním testování jsem zjišťoval počet porovnání a výměn v jednotlivých algoritmech vzhledem k různému množství vstupních dat. Obecně platí, že jedno porovnání se rovná jedné operaci a na jednu výměnu je zapotřebí operace tři. Přestože si lze z těchto údajů udělat jen hrubou představu o době třídění, velké rozdíly v počtu provedených operací mezi jednotlivými algoritmy jsou patrné na první pohled.



Obrázek 18: Graf - počet porovnání

Na málem testovacím vzorku nejsou rozdíly pochopitelně tolik patrné, ale již na vzorku o velikosti 1 000 se projevuje rozdíl mezi algoritmy s kvadratickou složitosti a algoritmy se složitostí logaritmickou.



Obrázek 19: Graf - počet výměn

U počtu výměn je rozdíl ještě propastnější. Bubblesort a Insertsort přestávají být na větším množství prvků použitelné. Ostatní algoritmy jsou poměrně vyrovnané. Překvapivě nízký počet výměn má Selectsort, to vychází z jeho principu, kdy v každé iteraci vymění právě jeden extrémní prvek. Bohužel tato výhoda je vykoupena vysokým počtem porovnávání. V celkovém výsledku se tedy s algoritmy s logaritmickou složitostí nemůže vyrovnat.

Dle seřazenosti pole

	$n = 1\,000$ náhodných		$n = 1\,000$ seřazených		$n = 1\,000$ antiseřazených	
	porovnání	**výměn**	**porovnání**	**výměn**	**porovnání**	**výměn**
Bubblesort	499 500	239 167	499 500	**0**	499 500	494 491
Insertsort	240166	239 167	999	**0**	495 490	494 491
Selectsort	499 500	999	499 500	999	499 500	999
Quicksort	14 541	4 447	501 501	1 000	97 121	46 353
Heapsort	24 569	9 430	23 283	8 211	26 678	12 072
Mergesort	10 685	9 976	10 685	9 976	10 685	9 976
Shellsort	11689	4599	7090	**0**	9 599	2 509
Combsort	18 010	3 064	18 010	**0**	18 010	1 258

Tabulka 7: Počet porovnání a výměn dle seřazenosti

Při testu na seřazeném a opačně seřazeném poli bylo cílem ověřit přirozenost jednotlivých třídících algoritmů. Z výsledků lze vyčíst, že algoritmy Bubblesort, Insertsort, Shellsort a Combsort svoji přirozenost při testování potvrdili. V průběhu testování u těchto algoritmů došlo pouze k porovnávání prvků bez nutnosti některý z nich přesunout na jinou pozici v

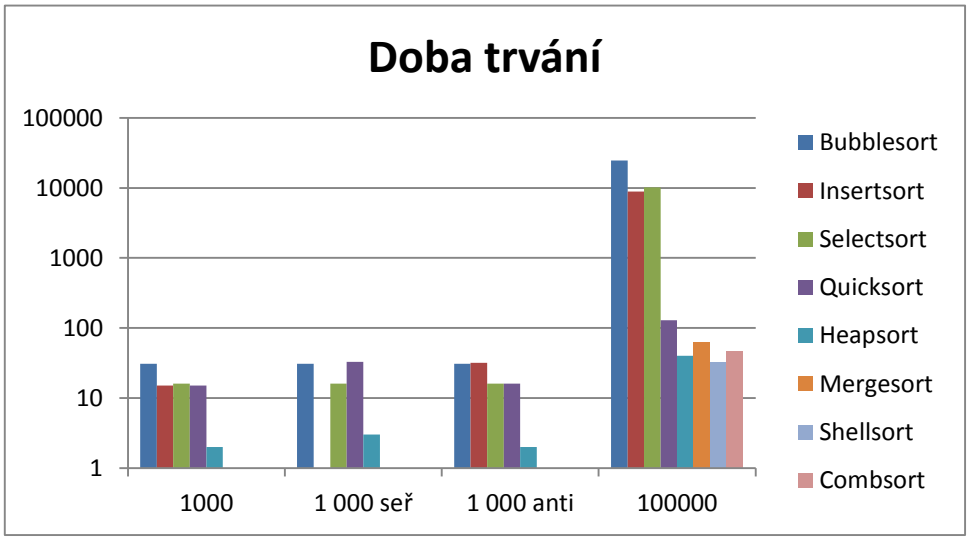
poli. V praxi je svojí přirozeností znám Insertsort, který se velmi často používá pro kontrolu, zda jsou vstupní data seřazena.

3. Srovnání podle doby trvání

$\Delta t\ [ms]$	$n = 10$	$n = 1\,000$	$n = 1\,000\ seř$	$n = 1\,000\ anti$	$n = 100\,000$
Bubblesort	0	31	31	31	24 576
Insertsort	0	15	0	32	8 880
Selectsort	0	16	16	16	10 126
Quicksort	0	15	33	16	129
Heapsort	0	2	3	2	40
Mergesort	0	0	0	0	63
Shellsort	0	0	0	0	32
Combsort	0	0	0	0	47

Tabulka 8: Doba trvání

Posledním výsledkem a porovnáním mého testování je celková doba trvání algoritmu řazení. Nula v tabulce vyjadřuje, že celý algoritmus proběhl rychleji než za 0,5 ms. Pro potřeby srovnání stačí posuzovat řád výsledků. U vstupních dat velikost $n = 10$ je porovnání nevypovídající, ale už při $n = 1000$ můžeme pozorovat, že zatímco algoritmy Mergesort, Shellsort a Combsort mají celkovou dobu trvaní maximálně stovky nanosekund, ostatní algoritmy mají celkovou dobu trvaní jednotky či dokonce desítky milisekund. To je řádově stonásobný (10^2) rozdíl.



Obrázek 20: Graf - doba trvání

Pochopitelně ještě razantnější rozdíl nalezneme u vstupních dat velikosti $n = 100\,000$. Nejrychlejší třídící algoritmy dosahují celkové doby v řádu desítek milisekund, zatímco pomalejší algoritmy již dosahují celkové doby v řádu desítek tisíc milisekund, tj. desítek sekund a to je již řádově tisícinásobný (10^3) rozdíl. V případě zvětšování velikosti vstupních dat by rozdíl nadále rostl.

5 Didaktická vhodnost algoritmů

Didaktickou vhodnost můžeme posuzovat podle různých faktorů, například dle obtížnosti, názornosti, používané metody atd. Výuku třídících algoritmů je možné realizovat v uceleném bloku jako samostatné učivo, nebo toto téma zakomponovat do výuky ostatních témat programování a tím výklad obohatit. Díky jasně viditelnému výstupu (žáci vidí seřazenou posloupnost) mohou vhodně zařazené úlohy působit i motivačně.

Základní řadící algoritmy Bubblesort, Selectsort a Insertsort jsou velmi vhodnými příklady pro výuku **polí a vnořování cyklů**. Princip řazení těchto algoritmů je velice jednoduchý a lze jej snadno názorně žákům předvést. Dále samotný zápis v programovacím jazyku je krátký, přehledný a je založený pouze na dvou vnořených for cyklech. Na těchto algoritmech lze příkladně demonstrovat vnořování cyklů a manipulaci s prvky pole.

Tyto základní algoritmy jsou natolik jednoduché a pochopitelné, že je vhodné při výuce dbát nejenom na schopnost algoritmy použit, ale i na porozumění průběhu a schopnost zapsat algoritmus v daném programovacím jazyce.

Pro výuku **rekurze**, tedy volání podprogramu na sebe, se nabízí Quicksort. Přestože u Quicksortu je zápis programu delší, jeho princip není složitý a problematiku rekurze presentuje dostatečně přehledně. Problematika ideální volby pivota může být námětem pro rozšiřující úlohu či diskusi.

Heapsort a Mergesort pro svoji obtížnost již nejsou pro počáteční výuku programování a třídění nejvhodnější. Pouze v případě pokročilejší výuky a zařazení struktury **binární haldy** do výuky lze pro ukázku jejího využití použít právě Heapsort.

Třídící algoritmy Shellsort a Combsort přímo vycházejí z principu Bubblesortu a Insertsortu. Proto je vhodné ve výuce tyto algoritmy využít pro rozšiřující či samostatné úlohy s cílem zaimplementovat do již známých algoritmů Bubblesort a Insertsort zmenšující se mezeru a tím eliminovat potíže těchto algoritmů.

Dalším tématem vhodným pro zakomponování třídění je práce se soubory. V rámci výuky **čtení ze souborů** a naopak **zápisu do souboru** se pro praktické příklady nabízí načítaná data setřídit a následně je setříděné vypsat do souboru nového. V tomto případě je na volbě vyučujících a znalostech žáků, které z algoritmů pro setřídění využijí.

Přestože se při třídění často automaticky uvažuje o třídění čísel v sestupném či vzestupném pořadí dle velikosti, je v rámci výuky **datového typu řetězec (*String*)** vhodné žákům prezentovat i možnosti třídění řetězců dle abecedy či alternativně dle délky řetězce.

Vypracovaný přehled algoritmů i příklady jejich zařazení do výuky může být inspirací pro vyučující, kteří třídění do výuky nezařazovali, nebo nevyužívali třídění ve cvičeních a úlohách.

Shrnutí výsledků testování třídících algoritmů

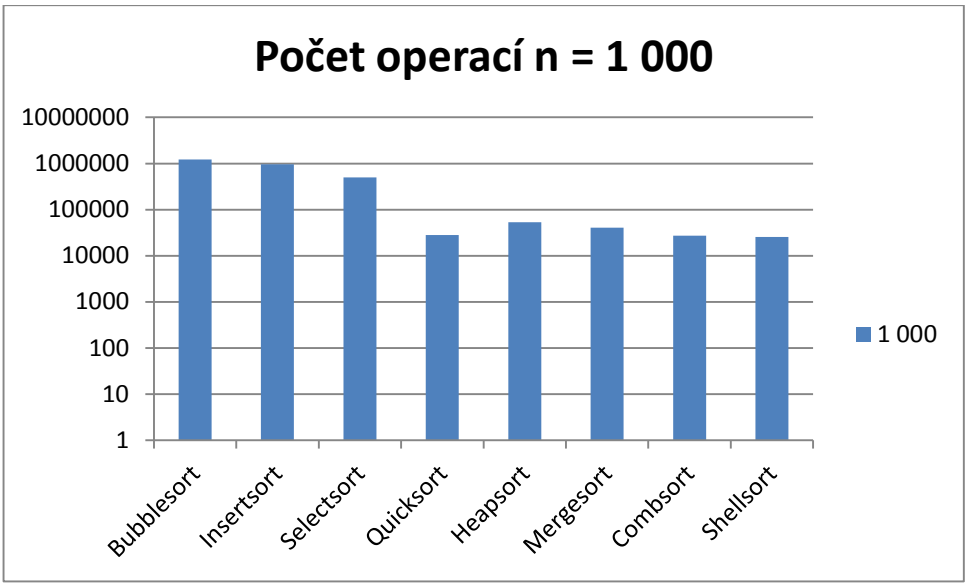
Výsledky, které jsme testováním získali, je vhodné porovnat a zhodnotit s teoretickými předpoklady a s testováním provedeným v jiných pracích.

V případě porovnávání výsledků testování s teoretickými předpoklady budeme pozornost věnovat třídě náročnosti. Algoritmy, které totožné třídy náročnosti by měli dle předpokladu vykazovat podobné výsledky.

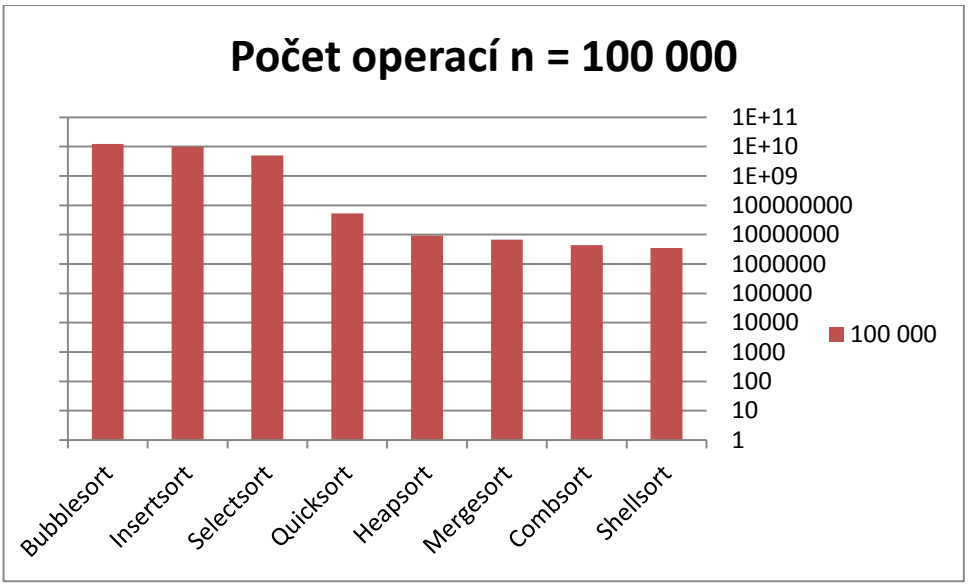
	Časová náročnost		$n = 1\,000$		$n = 100\,000$	
	avg θ	**max O**	**porovnání**	**výměn**	**porovnání**	**výměn**
Bubblesort	$O(n^2)$	$O(n^2)$	499 500	239 167	4 999 950 000	2 473 738 990
Insertsort	$O(n^2)$	$O(n^2)$	240 166	239 167	2 473 838 989	2 473 738 990
Selectsort	$O(n^2)$	$O(n^2)$	499 500	999	4 999 950 000	99 999
Quicksort	$O(n \log_2 n)$	$O(n^2)$	14 541	4 447	51 048 795	457 481
Heapsort	$O(n \log_2 n)$	$O(n \log_2 n)$	24 569	9 430	4 444 938	1 612 080
Mergesort	$O(n \log_2 n)$	$O(n \log_2 n)$	10 685	9 976	1 733 049	1 668 928
Combsort	$O(n \log n)$	$O(n \log n)$	18 010	3 064	3 397 054	320 274
Shellsort		$O(n^{4/3})$	11 689	4 599	1 859 535	551 189

Tabulka 9: Počet porovnání a výměn

Z tabulky nemusí být výsledek na první pohled patrný. Algoritmy se stejnou časovou náročností se v počtu provedených výměn a porovnání se také samozřejmě liší, ale rozdíl není veliký. Také velkou roli hraje konkrétní množina vstupních prvků a další vlastnosti algoritmů jako stabilita a přirozenost. Z grafu je rozdíl názornější:



Obrázek 21: Počet operací 1000



Obrázek 22:Počet operací 100 000

Bubblesort, Insertsort a Selectsort tvoří jasně vymezenou skupinu kvadratických algoritmů. U Quicksortu se potvrdil vliv volby pivota na výslednou efektivitu programu. Zatímco u 1 000 prvků patřil mezi algoritmy s nejmenším počtem operací, u druhého testu právě vlivem méně vhodné volby pivota za nejrychlejšími algoritmy zaostává. Následuje skupina tří lineárně logaritmických algoritmů - Heapsort, Mergesort, Shellsort a na závěr Shellsort s empiricky prokázanou polynomiální složitostí s koeficientem 4/3.

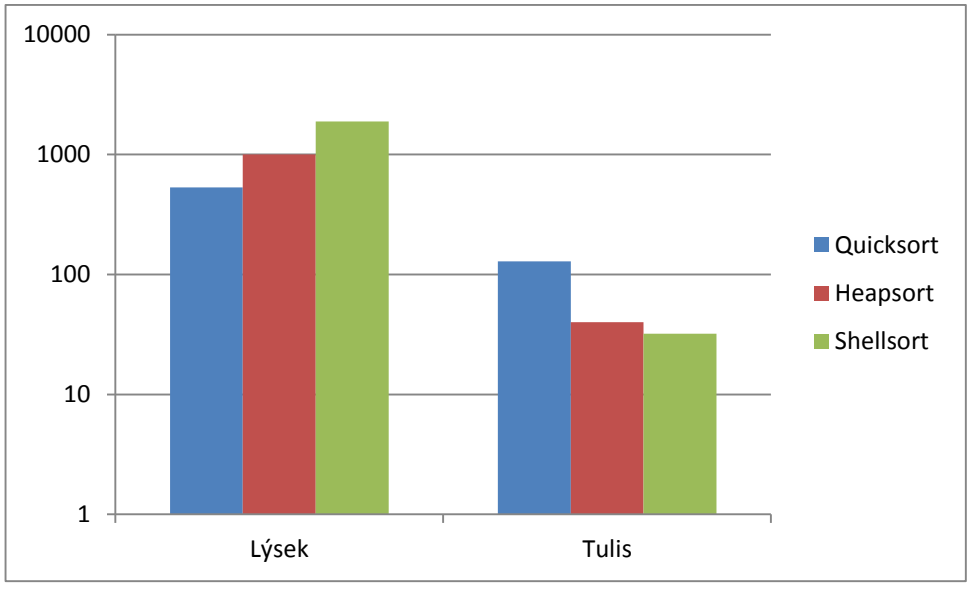
Lýsek (Lýsek, 2008) ve své bakalářské práci provádí podobné testování. Porovnáním obou testů můžeme dokázat, jak různé způsoby implementace mají zásadní vliv na efektivnost programu a tedy celkovou dobu trvání.

$\Delta t\ [ms]$	$n = 1\,000$	$n = 5\,000$	$n = 100\,000$
Bubblesort	5 388	22 172	
Insertsort	2 133	8 963	
Selectsort	120	470	
Quicksort	17	35	531
Heapsort	30	65	1 000
Shellsort	40	100	1 893

Tabulka 10: Srovnání doby trvání

Zpracováno podle: (Lýsek, 2008)

Přestože Lýsek pracuje s jinými velikostmi vstupních dat, lze pomocí grafů doby trvání jednotlivých algoritmů porovnat. Na množině vstupních dat o velikosti 100 000 prvků provedl testování pouze u rychlejších algoritmů – Quicksortu, Heapsortu a Shellsortu. Tento vzorek použijeme pro porovnání s našimi výsledky.



Zpracováno podle: (Lýsek, 2008)

U těchto tří algoritmů vidíme zásadní rozdíly mezi výsledky v naší práci a ve výsledcích Lýska. Tento rozpor snadno vysvětlíme po bližším posouzení implementace algoritmů. V případě Quicksortu rozdíl ve výsledcích způsobuje různá volba pivota. V případě Shellsortu efektivitu výrazně ovlivňuje volba mezery, respektive volba způsobu jejího zmenšování.

Závěr

Cílem této bakalářské práce bylo podat ucelený přehled nejpoužívanějších třídících algoritmů a v praktickém testování ověřit jejich vlastnosti.

Teoretická část byla členěna do 3 hlavních kapitol. První se věnuje úvodu do problematiky programování. Představuje syntaxi programovacího jazyku JAVA, způsoby deklarace a inicializace proměnných včetně přípustných datových typů. Dále popisuje řízení toku programu pomocí příkazů a větvení programu včetně cyklických příkazů.

Definováním základních pojmů se zabývá druhá kapitola. Popisuje historii vzniku pojmu *algoritmus*, jeho definici i nutné vlastnosti. Navíc předkládá možné způsoby zápisu. Dále se v kapitole podává definice samotného termínu *třídění.*

Třetí kapitola práce klasifikuje třídící algoritmy podle typu paměti, metody třídění, stabilnosti, přirozenosti a složitosti. Časová složitost, jako hlavní srovnávací parametr, je v kapitole popsána zevrubněji.

Hlavní částí práce je popis jednotlivých třídících algoritmů. Součástí popisu každého algoritmu je objasnění principu třídění a postupu včetně schématu, zobrazující průběh na vzorovém příkladu. Dále je proveden rozbor časové náročnosti a popsán kód zapsaný v programovacím jazyce JAVA.

V praktické části práce porovnávám třídící algoritmy Bubblesort, Selectsort, Insertsort, Quicksort, Heapsort, Mergesort, Shellsort a Combsort. Nejprve je provedeno obecné porovnání podle vlastností přirozenosti, stability a časové složitosti. Následně jsou na testovacích datech o velikosti 10, 1 000, 100 000 prvků jednotlivé vlastnosti ověřeny. Pro ověření přirozenosti algoritmů jsem do testování zařadil i třídění již setříděných a opačně setříděných vstupních dat.

Pro třídění malého počtu prvků jsou pro svoji jednoduchost vhodné základní třídící algoritmy třídy $O(n^2)$, pro početnější vstupní data je nutné použít rychlejší algoritmy, které dosahují třídy složitosti $O(n\,logn)$. Testováním byla dále potvrzena přirozenost algoritmů Insertsort, Bubblesort, Shellsort a Combsort. Tyto algoritmy při řazení již seřazeného pole neprováděli žádné výměny a tím byly na seřazeném poli nejrychlejší.

Jako nejrychlejší třídící algoritmus vyšel Shellsort. Tento vylepšený Bubblesort je velmi jednoduchý na implementaci a přesto dosahuje velice dobré výkonnosti. Navíc je i přirozený a lze jej tedy úspěšně využít i pro řazení částečně seřazených seznamů nebo k ověření seřazenosti.

Na závěr práce jsou jednotlivé algoritmy zhodnoceny po didaktické vhodnosti zařazení do výuky. Třídění je možné zařadit do výuky jako ucelené téma, nebo naopak jednotlivé třídící algoritmy představovat žákům jako vhodné příklady praktického využití právě probíraného učiva.

Citovaná literatura

Čápka, David. Heapsort. [Online] [Citace: 03. Únor 2015.] http://www.itnetwork.cz/algoritmus-heap-sort-trideni-cisel-podle-velikosti.

Kadlec, Václav. 2002. *Učíme se programovat v jazyce C.* Praha : Computer Press, 2002. ISBN 80-7226-715-9.

Keogh, Jim a Ken Davidson. 2006. *Datové struktury bez předchozích znalostí.* Brno : Computer Press, 2006. 80-251-0689-6.

Knuth, Donald E. 2008. *Umění programování.* Brno : Computer Press, 2008. ISBN 978-80-251-2025-5.

Lýsek, Jiří. 2008. *Algoritmy Třídění.* Brno : Vysoké učení technické v Brně, 2008.

Milková, Eva. 2008. *Algoritmy.* Praha : Alfa, 2008. ISBN 978-80-87197-10-3.

Prokop, Jiří. 2009. *Algoritmy v jazyku C a C++, praktický průvodce.* Praha : Grada, 2009. ISBN 978-80-247-2751-6.

Sedgewick, Robert. 2003. *Algoritmy v C.* Praha : SoftPress s.r.o, 2003. ISBN 80-86497-56-9.

Schildt, Herbert. 2001. *Java 2 - příručka programátora.* Brno : SoftPress, 2001. ISBN 80-86497-04-6.

Taufer, Ivan, Hrubina, Kamil a Taufer, Jan. 2001. *Algoritmy a algoritmizace Vývojové diagramy.* Hradec Králové : MAFY, 2001. ISBN 80-86148-49-1.

Töpfer, Pavel. 1995. *Algoritmy a programovací techniky.* Praha : Prometheus s.r.o., 1995. ISBN 80-85849-83-6.

Vaníček, Jiří. 2007. *Teoretické základy informatiky.* Praha : Alfa, 2007. ISBN 80-903962-4-1.

Virius, Miroslav. 2002. *C# pro zelenáče.* Praha : Neocortex, 2002. ISBN 80-86330-11-7.

Wróblewski, Piotr. 2004. *Algoritmy Datové struktury a programovací techniky.* Brno : Computer Press, 2004. ISBN 80-251-0343-9.

Seznam příloh

P1: Implementace Bubblesortu

P2: Implementace Selectsortu

P3: Implementace Insertsortu

P4: Implementace Quicksortu

P5: Implementace Heapsortu

P6: Implementace Mergesortu

P7: Implementace Shellsortu

P8: Implementace Combsortu

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruBS {
4
5    public static void main(String[] args) {
6
7      Scanner sc = new Scanner(System.in);
8      String nazevSouboru;
9
10     System.out.print("Zadej jméno souboru: ");
11     nazevSouboru = sc.next();
12
13     System.out.print("Zadej počet prvků: ");
14     int pocet = sc.nextInt();
15     int pole[] = new int [pocet];
16
17     try {
18       FileReader fr = new FileReader(nazevSouboru);
19       BufferedReader in = new BufferedReader(fr);
20
21
22       //plnění pole
23       for (int j=0; j < pole.length; j++) {
24         String radek = in.readLine();
25         int cislo = Integer.parseInt(radek);
26         pole[j] = cislo;
27       }
28
29       //bubble sort
30       long zmena = 0;
31       long porovnani = 0;
32       BubbleSort(pole, zmena, porovnani);
33
34       //vypis pole
35       System.out.println("Vypsáni souboru/pole");
36       for (int j=0; j < pole.length; j++)
37         System.out.print(pole[j] + " ");
38
39       fr.close();
40       in.close();
41     }
42     catch (IOException e) {
43       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
44     }
45   }
46
47
```

```
48    //--------------BubbleSort--------------
49    public static void BubbleSort(int[] pole, long zmena, long porovnani){
50      double startTime = System.currentTimeMillis();
51      for (int i = 0; i < pole.length - 1; i++) {
52        for (int j = 0; j < pole.length - i - 1; j++) {
53          porovnani = porovnani + 1;
54          if(pole[j] < pole[j+1]){
55            int pom = pole[j];
56            pole[j] = pole[j+1];
57            pole[j+1] = pom;
58            zmena = zmena + 1;
59          }
60        }
61      }
62      double stopTime = System.currentTimeMillis();
63      double time = stopTime - startTime;
64      System.out.println ("Time " +time);
65      System.out.println ("Zmena " +zmena);
66      System.out.println ("PocetPorovnani "+porovnani);
67    }
68  }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruSS {
4
5    public static void main(String[] args) {
6
7      Scanner sc = new Scanner(System.in);
8      String nazevSouboru;
9
10     System.out.print("Zadej jméno souboru: ");
11     nazevSouboru = sc.next();
12
13     System.out.print("Zadej počet prvků: ");
14     int pocet = sc.nextInt();
15     int pole[] = new int [pocet];
16
17     try {
18       FileReader fr = new FileReader(nazevSouboru);
19       BufferedReader in = new BufferedReader(fr);
20
21       //plnění pole
22       for (int j=0; j < pole.length; j++) {
23         String radek = in.readLine();
24         int cislo = Integer.parseInt(radek);
25         pole[j] = cislo;
26       }
27
28       //select sort
29       long zmena = 0;
30       long porovnani = 0;
31       Select(pole, zmena, porovnani);
32
33       //vypis pole
34       System.out.println("Vypsáni souboru/pole");
35       for (int j=0; j < pole.length; j++) {
36         System.out.print(pole[j] + " ");
37       }
38
39
40       fr.close();
41       in.close();
42     }
43     catch (IOException e) {
44       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
45     }
46   }
47
```

```
48    //--------------Select--------------
49    public static void Select(int[] pole, long zmena, long porovnani){
50      long startTime = System.currentTimeMillis();
51      for (int i = 0; i < pole.length - 1; i++) {
52        int maxIndex = i;          //index nejvetsiho prvku
53        for (int j = i + 1; j < pole.length; j++)
54        {
55          porovnani++;
56          if (pole[j] > pole[maxIndex])
57            maxIndex = j;
58        }
59        int pom = pole[i];
60        pole[i] = pole[maxIndex];
61        pole[maxIndex] = pom;
62      }
63
64      long stopTime = System.currentTimeMillis();
65      long time = stopTime - startTime;
66      System.out.println ("Time " +time);
67      System.out.println ("Zmena " +zmena);
68      System.out.println ("PocetPorovnani "+porovnani);
69    }
70  }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruIS {
4
5    public static void main(String[] args) {
6
7      Scanner sc = new Scanner(System.in);
8      String nazevSouboru;
9
10     System.out.print("Zadej jméno souboru: ");
11     nazevSouboru = sc.next();
12
13     System.out.print("Zadej počet prvků: ");
14     int pocet = sc.nextInt();
15     int pole[] = new int [pocet];
16
17     try {
18       FileReader fr = new FileReader(nazevSouboru);
19       BufferedReader in = new BufferedReader(fr);
20
21       //plnění pole
22       for (int j=0; j < pole.length; j++) {
23         String radek = in.readLine();
24         int cislo = Integer.parseInt(radek);
25         pole[j] = cislo;
26       }
27
28       //Insert sort
29       long zmena = 0;
30       long porovnani = 0;
31       Insert(pole, zmena, porovnani);
32
33       //vypis pole
34       System.out.println("Vypsáni souboru/pole");
35       for (int j=0; j < pole.length; j++) {
36         System.out.print(pole[j] + " ");
37       }
38
39       fr.close();
40       in.close();
41     }
42     catch (IOException e) {
43       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
44     }
45   }
46
```

```
47    //--------------Insert--------------
48    public static void Insert(int[] pole, long zmena, long porovnani){
49      long startTime = System.currentTimeMillis();
50      for (int i = 0; i < pole.length - 1; i++) {
51        int j = i + 1;
52        int x = pole[j];
53        while (j > 0 && x > pole[j-1]) {
54          pole[j] = pole[j-1];
55          j--;
56          porovnani++;
57          zmena++;
58        }
59        pole[j] = x;
60        porovnani++;
61      }
62      long stopTime = System.currentTimeMillis();
63      long time = stopTime - startTime;
64      System.out.println ("Time " +time);
65      System.out.println ("Zmena " +zmena);
66      System.out.println ("PocetPorovnani "+porovnani);
67    }
68  }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruQS {
4
5    static int porovnani = 0;
6    static int zmena = 0;
7
8    public static void main(String[] args) {
9
10     Scanner sc = new Scanner(System.in);
11     String nazevSouboru;
12
13     System.out.print("Zadej jméno souboru: ");
14     nazevSouboru = sc.next();
15
16     System.out.print("Zadej počet prvků: ");
17     int pocet = sc.nextInt();
18     int pole[] = new int [pocet];
19
20     try {
21       FileReader fr = new FileReader(nazevSouboru);
22       BufferedReader in = new BufferedReader(fr);
23
24       //plnění pole
25       for (int j=0; j < pole.length; j++) {
26         String radek = in.readLine();
27         int cislo = Integer.parseInt(radek);
28         pole[j] = cislo;
29       }
30
31       //Quick sort
32       long startTime = System.currentTimeMillis();
33
34       quicksort(pole,0, pocet);
35
36       long stopTime = System.currentTimeMillis();
37       long time = stopTime - startTime;
38       System.out.println ("Time " +time);
39       System.out.println ("Zmena " +zmena);
40       System.out.println ("PocetPorovnani "+ porovnani);
41
42       //vypis pole
43       System.out.println("Vypsáni souboru/pole");
44       for (int j=0; j < pole.length; j++) {
45         System.out.print(pole[j] + " ");
46       }
47
48       fr.close();
49       in.close();
50     }
51     catch (IOException e) {
52       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
53     }
54   }
```

```
55
56    //--------------Quick--------------
57    public static void quicksort(int[] pole , int levy , int pravy){
58
59      porovnani++;
60      if(levy < pravy)
61      {
62        int piv = levy; //pivot
63        for(int i = levy + 1; i < pravy; i++){
64          porovnani++;
65          if(pole[i] > pole[levy])
66            swap(pole, i, ++piv);
67        }
68
69        swap(pole,  levy, piv);
70        quicksort(pole, levy, piv);
71        quicksort(pole, piv + 1, pravy);
72      }
73    }
74
75    private static void swap(int[] pole, int levy, int pravy){
76      int a = pole[pravy];
77      pole[pravy] = pole[levy];
78      pole[levy] = a;
79      zmena++;
80    }
81  }
82
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruHS {
4
5    static int porovnani = 0;
6    static int zmena = 0;
7
8    public static void main(String[] args) {
9
10     Scanner sc = new Scanner(System.in);
11     String nazevSouboru;
12
13     System.out.print("Zadej jméno souboru: ");
14     nazevSouboru = sc.next();
15
16     System.out.print("Zadej počet prvků: ");
17     int pocet = sc.nextInt();
18     int pole[] = new int [pocet];
19
20     try {
21       FileReader fr = new FileReader(nazevSouboru);
22       BufferedReader in = new BufferedReader(fr);
23
24       //plnění pole
25       for (int j=0; j < pole.length; j++) {
26         String radek = in.readLine();
27         int cislo = Integer.parseInt(radek);
28         pole[j] = cislo;
29       }
30
31       //heap sort
32       long startTime = System.currentTimeMillis();
33
34       heapsort(pole);
35
36       long stopTime = System.currentTimeMillis();
37       long time = stopTime - startTime;
38       System.out.println ("Time " +time);
39       System.out.println ("Zmena " +zmena);
40       System.out.println ("PocetPorovnani "+ porovnani);
41
42       //vypis pole
43       System.out.println("Vypsáni souboru/pole");
44       for (int j=0; j < pole.length; j++) {
45         System.out.print(pole[j] + " ");
46       }
47
48       fr.close();
49       in.close();
50     }
51     catch (IOException e) {
52       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
53     }
54   }
```

```
55
56    //--------------Heapsort--------------
57   //oprava haldy nahoru
58    public static void up(int[] list, int i) {
59      int child = i; //ulozim syna
60      int parrent, temp;
61      while (child != 0) {
62        parrent = (child - 1) / 2; //otec
63        porovnani++;
64        if (list[parrent] < list[child]) { //detekce chyby
65          zmena++;
66          temp = list[parrent]; //prohozeni syna s otcem
67          list[parrent] = list[child];
68          list[child] = temp;
69          child = parrent; //novy syn
70        }
71        else
72          return;
73      }
74    }
75
76  //oprava haldy dolu
77    public static void down(int[] list, int last) {
78      int parrent = 0;
79      int child, temp;
80      while (parrent * 2 + 1 <= last) {
81        child = parrent * 2 + 1;
82        // pokud je vybran mensi syn
83        porovnani++;
84        porovnani++;
85        if ((child < last) && (list[child] < list[child + 1]))
86          child++; //vybereme toho vetsiho
87        porovnani++;
88        if (list[parrent] < list[child]) { //detekce chyby
89          zmena++;
90          temp = list[parrent]; //prohozeni syna s otcem
91          list[parrent] = list[child];
92          list[child] = temp;
93          parrent = child; //novy otec
94        }
95        else
96          return;
97      }
98    }
99
100 // postaveni haldy z pole
101   public static void heapify(int[] list) {
102     for (int i = 1; i < list.length; i++)
103       up(list, i);
104   }
```

```
105
106 // samotne trideni
107   public static void heapsort(int[] list) {
108     heapify(list);
109     int index = list.length - 1; // posledni prvek
110     int temp;
111     while (index > 0) {
112       zmena++;
113       temp = list[0]; // prohození posledního prvku s maximem
114       list[0] = list[index];
115       list[index] = temp;
116       index -= 1; //nastaveni noveho posledniho prvku
117       down(list, index);
118     }
119   }
120 }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruMS {
4
5    static int porovnani = 0;
6    static int zmena = 0;
7
8    public static void main(String[] args) {
9
10     Scanner sc = new Scanner(System.in);
11     String nazevSouboru;
12
13     System.out.print("Zadej jméno souboru: ");
14     nazevSouboru = sc.next();
15
16     System.out.print("Zadej počet prvků: ");
17     int pocet = sc.nextInt();
18     int pole[] = new int [pocet];
19
20     try {
21       FileReader fr = new FileReader(nazevSouboru);
22       BufferedReader in = new BufferedReader(fr);
23
24       //plnění pole
25       for (int j=0; j < pole.length; j++) {
26         String radek = in.readLine();
27         int cislo = Integer.parseInt(radek);
28         pole[j] = cislo;
29       }
30
31       //merge sort
32
33       double startTime = System.currentTimeMillis();
34
35       mergeSort(pole);
36
37       double stopTime = System.currentTimeMillis();
38       double time = stopTime - startTime;
39       System.out.println ("Time " +time);
40       System.out.println ("Zmena " +zmena);
41       System.out.println ("PocetPorovnani "+porovnani);
42
43       // vypis pole
44       System.out.println("Vypsáni souboru/pole");
45       for (int j=0; j < pole.length; j++) {
46         System.out.print(pole[j] + " ");
47       }
48       fr.close();
49       in.close();
50     }
51     catch (IOException e) {
52       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
53     }
54   }
```

```
55
56    //--------------MergeSort--------------
57    public static void merge(int[] pole, int[] left, int[] right) {
58      int i = 0;
59      int j = 0;
60      // dokud nevyjedeme z jednoho z poli
61      porovnani++; porovnani++;
62      while ((i < left.length) && (j < right.length)) {
63        // dosazeni toho mensiho prvku z obou poli a posunuti indexu
64        porovnani++;
65        if (left[i] > right[j]) {
66          pole[i + j] = left[i];
67          i++;
68          zmena++;
69        }
70        else {
71          pole[i + j] = right[j];
72          j++;
73          zmena++;
74        }
75      }
76      // doliti zbytku z nevyprazdneneho pole
77      porovnani++;
78      if (i < left.length) {
79        while (i < left.length) {
80          pole[i + j] = left[i];
81          i++;
82          zmena++;
83        }
84      }
85      else {
86        while (j < right.length) {
87          pole[i + j] = right[j];
88          j++;
89          zmena++;
90        }
91      }
92    }
93
```

```
94  // samotne trideni
95    public static void mergeSort(int[] pole) {
96      if (pole.length <= 1) //podminka rekurze
97        return ;
98      int center = pole.length / 2; //stred pole
99      int[] left = new int[center]; //vytvoreni a naplneni leveho pole
100     for (int i = 0; i < center; i++)
101     {
102       left[i] = pole[i];
103     }
104     int[] right = new int[pole.length - center]; //vytvoreni a naplneni
105     for (int i = center; i < pole.length; i++)  //vytvoreni a naplneni
106     {
107       right[i - center] = pole[i];
108     }
109     mergeSort(left); // rekurzivni zavolani na obe nova pole
110     mergeSort(right);
111     merge(pole, left, right); //sliti poli
112   }
113 }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruShS {
4
5    public static void main(String[] args) {
6
7      Scanner sc = new Scanner(System.in);
8      String nazevSouboru;
9
10     System.out.print("Zadej jméno souboru: ");
11     nazevSouboru = sc.next();
12
13     System.out.print("Zadej počet prvků: ");
14     int pocet = sc.nextInt();
15     int pole[] = new int [pocet];
16
17     try {
18       FileReader fr = new FileReader(nazevSouboru);
19       BufferedReader in = new BufferedReader(fr);
20
21       //plnění pole
22       for (int j=0; j < pole.length; j++) {
23         String radek = in.readLine();
24         int cislo = Integer.parseInt(radek);
25         pole[j] = cislo;
26       }
27
28       //shell sort
29       long zmena = 0;
30       long porovnani = 0;
31       BubbleSort(pole, zmena, porovnani);
32
33      // vypis pole
34       System.out.println("Vypsáni souboru/pole");
35       for (int j=0; j < pole.length; j++) {
36         System.out.print(pole[j] + " ");
37       }
38
39       fr.close();
40       in.close();
41     }
42     catch (IOException e) {
43       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
44     }
45   }
46
```

```
47    //--------------ShellSort--------------
48    public static void BubbleSort(int[] pole, long zmena, long porovnani){
49      long startTime = System.currentTimeMillis();
50      int gap = pole.length / 2; //deleni mezery 2
51      while (gap > 0) {
52        for (int i = 0; i < pole.length - gap; i++) { // insertion sort
53          int j = i + gap;
54          int tmp = pole[j];
55          while (j >= gap && tmp > pole[j - gap]) {
56            pole[j] = pole[j - gap];
57            j -= gap;
58            porovnani++;
59            zmena++;
60          }
61          pole[j] = tmp;
62          porovnani++;
63        }
64        if (gap == 2)  //zmena velikosti mezery
65          gap = 1;
66        else
67          gap /= 2.2; //zmena velikosti mezery
68      }
69      long stopTime = System.currentTimeMillis();
70      long time = stopTime - startTime;
71      System.out.println ("Time " +startTime);
72      System.out.println ("Time " +stopTime);
73      System.out.println ("Time " +time);
74      System.out.println ("Zmena " +zmena);
75      System.out.println ("PocetPorovnani "+porovnani);
76    }
77  }
```

```
1  import java.util.*;
2  import java.io.*;
3  class CteniSouboruCS {
4
5    public static void main(String[] args) {
6
7      Scanner sc = new Scanner(System.in);
8      String nazevSouboru;
9
10     System.out.print("Zadej jméno souboru: ");
11     nazevSouboru = sc.next();
12
13     System.out.print("Zadej počet prvků: ");
14     int pocet = sc.nextInt();
15     int pole[] = new int [pocet];
16
17     try {
18       FileReader fr = new FileReader(nazevSouboru);
19       BufferedReader in = new BufferedReader(fr);
20
21       //plnění pole
22       for (int j=0; j < pole.length; j++) {
23         String radek = in.readLine();
24         int cislo = Integer.parseInt(radek);
25         pole[j] = cislo;
26       }
27
28       //CombSort
29       long zmena = 0;
30       long porovnani = 0;
31       BubbleSort(pole, zmena, porovnani);
32
33       //vypis pole
34       System.out.println("Vypsáni souboru/pole");
35       for (int j=0; j < pole.length; j++) {
36         System.out.print(pole[j] + " ");
37       }
38
39       fr.close();
40       in.close();
41     }
42     catch (IOException e) {
43       System.out.println("Soubor " + nazevSouboru + "nelze otevřít");
44     }
45   }
46
```

```
46
47    //--------------CombSort--------------
48    public static void BubbleSort(int[] pole, long zmena, long porovnani){
49      long startTime = System.currentTimeMillis();
50      int mezera = pole.length;
51      while (mezera != 1) {
52        mezera /= 1.33; //zmenseni mezery o 4/3
53
54        for (int i = 0; i + mezera < pole.length; i++) {
55          porovnani++;
56          if (pole[i] < pole[i + mezera]) {
57            int tmp = pole[i];
58            pole[i] = pole[i + mezera];
59            pole[i + mezera] = tmp;
60            zmena++;
61          }
62        }
63      }
64      long stopTime = System.currentTimeMillis();
65      long time = stopTime - startTime;
66      System.out.println ("Time " +startTime);
67      System.out.println ("Time " +stopTime);
68      System.out.println ("Time " +time);
69      System.out.println ("Zmena " +zmena);
70      System.out.println ("PocetPorovnani "+porovnani);
71    }
72  }
```