VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

ALGORITMY TŘÍDĚNÍ

SORTING ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JAKUB SCHWARZ
AUTHOR

VEDOUCÍ PRÁCE — RNDr. JIŘÍ DVOŘÁK, CSc.
SUPERVISOR

BRNO 2011

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2010/2011

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Jakub Schwarz

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

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

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

Algoritmy třídění

v anglickém jazyce:

Sorting algorithms

Stručná charakteristika problematiky úkolu:

Třídění je proces uspořádání objektů podle nějakého kritéria. Jeho účelem je usnadnit pozdější vyhledávání těchto objektů.

Cíle bakalářské práce:

1. Charakterizovat problematiku třídění.
2. Popsat vybrané principy a algoritmy třídění.
3. Na základě údajů z literatury (a případně vlastních experimentů) provést jejich srovnání.

Seznam odborné literatury:

WIRTH, N. Algoritmy a štruktúry údajov. Bratislava, ALFA 1988.
HONZÍK, J. a kol. Programovací techniky. Skripta. VUT v Brně, 1985.

Vedoucí bakalářské práce: RNDr. Jiří Dvořák, CSc.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2010/2011.

V Brně, dne 18.11.2010

L.S.

Ing. Jan Roupec, Ph.D. Ředitel ústavu	prof. RNDr. Miroslav Doupovec, CSc. Děkan fakulty

Abstrakt

Při obrovských objemech dat, které se během výrobních procesů zpracovávají, je snadná orientace a hledání v nich zcela zásadní. Správné a rychlé třídění dat je jednou z nejdůležitějších činností při jejich zpracování.

Cílem této bakalářské práce je provést rešerši algoritmů třídění. K tomuto cíli budou vymezeny základní pojmy v oblasti třídění a popsáno rozdělení třídicích algoritmů podle různých kritérií. U každého z vybraných algoritmů vnitřního třídění polí bude analyzován princip třídění a proveden rozbor časové efektivnosti. Výsledky budou ověřeny experimentálním programem.

Abstract

Enormous volume of data, which has been processed during production operations, requires easy orientation and quick searching. Exact search in real time is essential for success. Correct and quick data-sorting is one of the most important activity during data-processing.

The goal of this bachelor's thesis is background research of sorting algorithms. To achieve the goal, basic concepts of sorting theory will be described (determined) and there will be division of sorting algorithms by different criterions. For each algorithm of internal array sorting, sorting principles will be described and analysis of efficiency will be provided. The results will be verified by experimental program.

Klíčová slova

Algoritmus, třídění, pole, složitost

Keywords

Algorithm, sorting, array, complexity

Poděkování

Děkuji tímto RNDr. Jiřímu Dvořákovi, CSc. za mnoho užitečných rad a připomínek při zpracovávání bakalářské práce, Ing. Stanislavovi Věchetovi, Ph.D. za výbornou výuku programování v prostředí C#, přítelkyni Zuzce za trpělivost a kamarádovi Žajdovi za pomoc při dokončení programu.

Prohlášení

Prohlašuji, že jsem bakalářskou práci na téma algoritmy třídění vypracoval samostatně s použitím pramenů uvedených v seznamu použité literatury.

V Brně 16. května 2011 ……

Podpis autora

Obsah:

1 Úvod

Tříděním rozumíme proces postupného přeuspořádávání daného pole objektů v předem daném specifickém pořadí s cílem ulehčit pozdější vyhledávání a orientaci v poli. Třídění zasahuje do všech možných druhů databází, kde je potřeba uchovávaná data vyhledávat a vybírat podle nějakých klíčů. Třídění je využíváno např. při tvorbě telefonních seznamů a při práci se všemi druhy registrů zboží. Třídění je velmi důležitou a základní činností při zpracovávaní údajů.

V následujícím textu bude pojednáno o základech problematiky třídění, bude provedeno rozdělení třídicích algoritmů podle charakteristických parametrů a budou zavedena kritéria pro porovnávání jednotlivých algoritmů.

Samotný popis jednotlivých metod se bude týkat pouze algoritmů určených pro vnitřní třídění polí. Jednotlivé algoritmy budou implementovány do experimentálního programu, který bude mít za úkol vyhodnocovat a porovnávat časovou náročnost jednotlivých metod.

2 Problematika třídění

Mezi základní pojmy v procesech třídění patří tyto[1]:

- Třídění je definováno jako rozdělování údajů na skupiny údajů se stejnými vlastnostmi.

- Uspořádání podle klíčů je definováno jako seřazení údajů podle prvků (klíčů) lineárně uspořádané množiny.

- Řazení je definováno jako uspořádání údajů podle relace lineárního uspořádání.

- Slučování je definováno jako vytváření souboru sjednocením několika souborů.

- Setřídění je definováno jako vytváření souboru sjednocením několika souborů, jejichž údaje jsou seřazeny podle téže relace uspořádání se zachováním této relace.

- Stabilita třídění je definována tak, že pokud relativní pořadí prvků se stejným klíčem zůstává v procesu třídění nezměněné, je třídění stabilní. Stability se využívá v případech, když jsou prvky už uspořádané podle jiných sekundárních klíčů.

2.1 Rozdělení algoritmů třídění

Metody třídění lze rozdělit podle základních parametrů a principů takto[1]:

A) Dělení podle typu paměti, v níž je tříděná struktura uložena:

- *vnitřní třídění* neboli metody třídění polí předpokládá uložení v operační paměti a přímý (nesekvenční) přístup k položkám struktury, třídění probíhá pouze s jedním polem.

-*vnější třídění* neboli metody třídění souborů předpokládá sekvenční přístup k položkám seřazované struktury. Zvláštní skupinu mohou tvořit struktury s indexsekvenčním přístupem (implementované na magnetických discích), u nichž metody třídění často kombinují principy vnitřního a vnějšího třídění.

B) Dělení podle typu procesoru:

- *sériové* na sériovém procesoru se může každá následující operace algoritmu provést až po dokončení předcházející.

- *paralelní* paralelní procesor umožňuje současný průběh více operací, čímž urychluje průběh třídění.

C) Dělení na přímé a nepřímé metody

- *přímé metody* využívají daného principu třídění v nejjednodušší podobě. Jsou jednoduché, ale mají vysokou časovou náročnost. Jsou vhodné pro malý počet seřazovaných položek.

- *nepřímé metody* vycházejí z týchž principů jako metody přímé, ale vylepšují základní princip různými programovacími metodami. Jsou složitější pro zápis algoritmu i pro jeho pochopení, ale mají nižší časovou náročnost. Jsou vhodné pro větší počty seřazovaných položek.

D) Dělení podle základního principu třídění

- *princip výběru:* metoda přesouvá postupně největší (nejmenší) prvek z tříděné množiny do výstupní lineární struktury.

- *princip vkládání:* metoda zařazuje do setříděné výstupní lineární struktury postupně všechny prvky seřazované množiny (v libovolném pořadí).

- *princip rozdělování:* metoda rozděluje postupně všechny množiny a podmnožiny na dvě další podmnožiny tak, že všechny prvky jedné podmnožiny jsou menší než prvky druhé podmnožiny.

- *princip setřídění:* metoda sjednocuje seřazené podmnožiny do větších seřazených podmnožin.

- *jiné principy*: jedná se o méně sourodou skupinu různých principů nebo kombinaci základních principů.

2.2 Hodnocení časové náročnosti třídicích algoritmů

Metody třídění polí by měly především úsporně využívat paměť, kterou mají k dispozici. Všechny permutace při přeuspořádávání prvků pole by se měly vykonávat bez použití pomocné paměti. Efektivnost metody se měří pomocí časové náročnosti vyjádřené počtem potřebných operací pro utřídění pole, především počtem potřebných porovnání klíčů C a počtem přesunů prvků M. Tyto počty jsou funkcí počtu prvků v poli n, které se mají setřídit. Při vyhodnocování časové složitosti algoritmů se každému příkazu přiřadí odpovídající počet časových jednotek potřebných k provedení příkazu a počet jednotek paměti, které tento příkaz využije ke svému provedení. S rostoucí výkonností počítačů se úměrně zvětšuje i rozsah řešených problémů a tedy i rozsah vnitřně roztříditelných polí. Nejlepší třídící algoritmy dosahují složitosti $n \cdot \log n$, zatímco jednoduché metody mívají složitost n^2. Pro názorné porovnání časové složitosti jednotlivých algoritmů zavedeme reprezentanty o různých složitostech takto $A_1 = 1000n$, $A_2 = 100n \cdot log\ n$, $A_3 = 10n^2$, $A_4 = n^3$, $A_5 = 2^n$. Při těchto předpokladech bude algoritmus A_5 nejvhodnější pro malá pole s počtem prvků $n \leq 9$, A_3 pro $10 \leq n \leq 58$, A_2 pro $59 \leq n \leq 1024$ a A_1 pro $n \geq 1024$ [1].

algoritmus	složitost	maximální rozsah problému (n)			vliv 10x zvýšení výkonu na rozsah	
		1 s	1 min	1 hod	rozsah před	rozsah po
A_1	n	1000	$6 \cdot 10^4$	$3{,}6 \cdot 10^6$	S_1	$10 \cdot S_1$
A_2	n·log n	140	4893	$2 \cdot 10^5$	S_2	$\approx 10 \cdot S_2$
A_3	n^2	31	244	1897	S_3	$3{,}16 \cdot S_3$
A_4	n^3	10	39	153	S_4	$2{,}15 \cdot S_4$
A_5	2^n	9	15	21	S_5	$S_5+3{,}3$

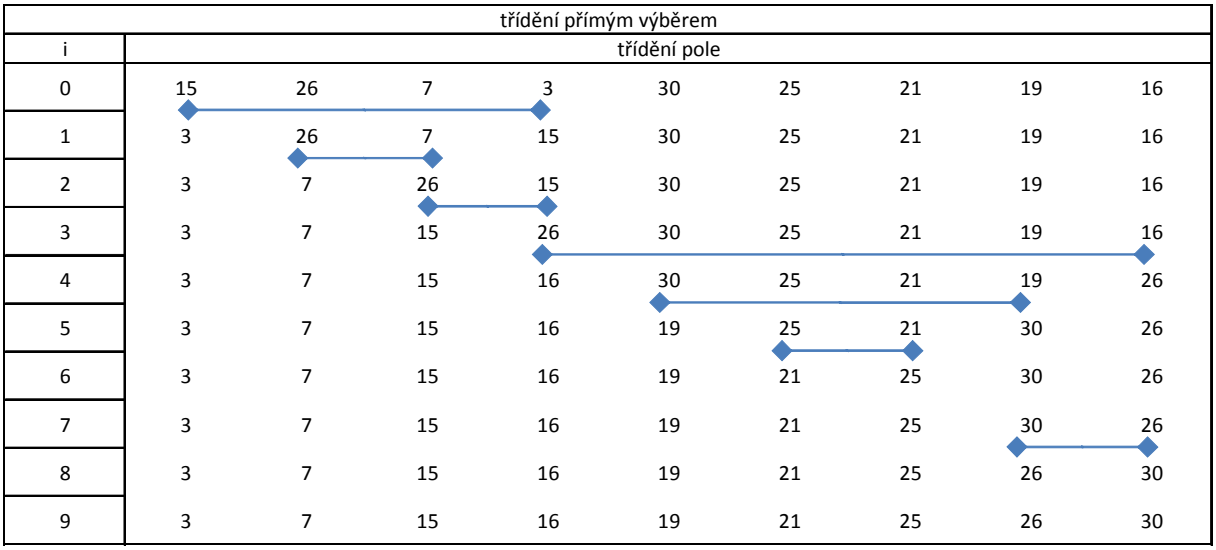
Tab. 1 Porovnání časové a prostorové náročnosti [1]

3 Algoritmy vnitřního třídění polí

Následuje popis vybraných algoritmů vnitřního třídění polí. Mezi zástupce jednoduchých metod patří metody přímého výběru, přímého vkládání a bublinové třídění. Jako mezistupeň k efektivním metodám bude uveden algoritmus třídění se zmenšováním kroku. A z efektivních metod budou analyzovány metoda stromového třídění a metoda třídění pomocí rozdělování.

3.1 Metoda přímého výběru

Princip metody je velmi jednoduchý, tříděné pole je rozděleno na dvě části, na první utříděnou s indexy 1..*i*-1 a na druhou neutříděnou s indexy *i*..*n*, z níž se vybírají nejmenší prvky a řadí se na konec první části tak, že po nalezení nejmenšího prvku se tento prvek vymění s prvním prvkem neuspořádané části a utříděná část se rozroste právě o tento prvek. Proces třídění začíná s první částí prázdnou a končí porovnáním a utříděním prvků s indexy *n-1* a *n*.



Obr 1. Schéma třídění přímým výběrem

```
classprimyvyber : TridiciAlgoritmus
    {
        publicint[] prvy(int[] mojepole)
           {
              for (int i = 0; i <mojepole.Length; i++) // vnější cyklus třídění
                 {
                    int k = i;                          // index polohy pomocné proměnné
                    int pom = mojepole[i];              // přiřazení pomocné proměnné
                    for (int j = i + 1; j <mojepole.Length; j++)
                       {                                // vnitřní cyklus třídění
                          if (mojepole[j] < pom)        // porovnání prvků
                             {
                                   pom = mojepole[j];   // výběr nejmenšího prvku
                                   k = j;               // změna polohy pomocné proměnné
                             }
                             this.pocetporovnavi = this.pocetporovnavi + 1;
                       }
                    mojepole[k] = mojepole[i];          // výměna prvků
                    mojepole[i] = pom;
```

```
                this.pocetvymen = this.pocetvymen + 1;
            }
        Return mojepole;
        }
```

Program 1 algoritmus přímého výběru

Metoda přímého výběru patří mezi jednoduché algoritmy s časovou náročností n^2. Je nestabilní, protože může dojít ke změně relativního pořadí dvou stejně velkých prvků při výměně nejmenšího prvku za prvek s nejnižším indexem v neutříděném podpoli. Metoda pracuje bez využívání pomocné paměti, tj. in situ.[1]

Počet porovnání mezi prvky zde nezávisí na počátečním uspořádání prvků v poli. Proto lze říci, že střední hodnota počtů porovnání C v závislosti na *n* je [2]:

$$C \cong \frac{1}{2} \cdot (n^2 - n)$$

Střední hodnotu *M* počtu výměn lze těžko určit, protože výrazně závisí na rozložení prvků v netříděné části pole. Čím více se budou prvky s největší hodnotou vyskytovat na počátečních místech neuspořádaného podpole, tím více se hodnota *M* se bude zvětšovat. Při náhodném rozložení prvků v tříděném poli lze říci, že aproximovaná střední hodnota *M* je[2]:

$$M \cong n \cdot (\ln n + 0{,}577216)$$

3.2 Metoda přímého vkládání

Metoda opět využívá rozdělení tříděného pole na dvě části. Z první zdrojové neuspořádané podčásti se vybírá k porovnání vždy první člen. Neuspořádané podpole má indexy *i* až *n*. Vybraný prvek se nejdříve porovnává s prvkem o indexu *i*-1, a pak s každým dalším prvkem s nižším indexem dokud se nedostane na první pozici nebo prvek s nižším indexem je nižší, nebo roven vybranému prvku. Tento proces se opakuje, dokud není zdrojové podpole prázdné. Algoritmus začíná porovnávat prvky od indexu $i = 2$ a v každém kroku procesu se *i* zvětší o jedničku.

třídění přímým vkládáním									
i	třídění pole								
0	15	26	7	3	30	25	21	19	16
2	15	26	7	3	30	25	21	19	16
3	7	15	26	3	30	25	21	19	16
4	3	7	15	26	30	25	21	19	16
5	3	7	15	26	30	25	21	19	16
6	3	7	15	25	26	30	21	19	16
7	3	7	15	21	25	26	30	19	16
8	3	7	15	19	21	25	26	30	16
9	3	7	15	16	19	21	25	26	30

Obr 2. Schéma třídění přímým vkládáním

```
class primevkladani : TridiciAlgoritmus
```

```
{
    public int[] primevkl(int[] mojepole)
    {
        int podminka = 2;                          // podmínka vnitřního cyklu
        for (int k = 1; k < mojepole.Length; k++) // vnější třídící cyklus
        {
            int pom = mojepole[k];                 // přiřazení pomocné proměnné
            int j = k - 1;                         // nastavení počátku
                                                   prohledávání
            while (podminka > 1)                   // vnitřní cyklus prohledávání
            {
                if (pom < mojepole[j])             // porovnání prvků
                {
                    mojepole[j + 1] = mojepole[j]; // posun porovnávaných prvků
                    this.pocetvymen = this.pocetvymen + 1;
                    j = j - 1;
                    if (j < 0)                     // kontrola okraje pole
                    {podminka = 0;}
                }
                else
                {podminka = 0;}                    // prvek dosáhl své pozice
                this.pocetporovnavi = this.pocetporovnavi + 1;
            }
            mojepole[j + 1] = pom;                 // utřídění prvku
            podminka = 2;
        }
        return mojepole;
    }
```

Program 2 algoritmus přímého vkládání

Rovněž metoda přímého vkládání patří mezi jednoduché algoritmy s časovou a prostorovou náročností n^2. Metoda je stabilní, nemůže dojít k výměně dvou prvků se stejnou hodnotou a tím k narušení relativního pořadí prvků během třídění. Metoda se chová přirozeně a pracuje bez využívání pomocné paměti, tj. in situ [1].

Celkový počet porovnání opět závisí na uspořádání pole před tříděním. Nejmenší počet nastává v případě, když je pole utříděné. Nejhorší případ nastane při opačném uspořádání pole, tedy když je pole uspořádáno od největšího prvku k nejmenšímu. Hodnoty počtu porovnání pro nejlepší, střední a nejhorší případ jsou[2]:

$$C_{min} = n - 1$$

$$C_{střed} \cong \frac{1}{4} \cdot (n^2 + 3n - 4)$$

$$C_{max} = \frac{1}{2} \cdot (n^2 + n) - 1$$

Celkový počet výměn závisí taktéž na uspořádání pole před tříděním, minimální a maximální případy počtu výměn nastávají při stejném uspořádání jako minimální a maximální případy počtu porovnání. Hodnoty počtu výměn pro nejlepší, střední a nejhorší případ jsou[2]:

$$M_{min} = 3 \cdot (n\text{-}1)$$

$$M_{střed} \cong \frac{1}{4} \cdot (n^2 + 11n - 12)$$

$$M_{max} = \frac{1}{2} \cdot (n^2 + 5n - 6)$$

3.3 Metoda bublinového třídění

Základní princip metody tkví opět v porovnávání vedle sebe stojících prvků. Odlišuje se však ve způsobu hledání minima a jeho zařazení na správné místo v utříděném poli. Algoritmus rozděluje pole na dvě části. První podpole je seřazené a na začátku třídění prázdné. Druhé neroztříděné podpole se při hledání minima prochází od konce do prvního neutříděného prvku a porovnávají se při tom každé dva sousední prvky. Tímto způsobem se nejmenší prvek dostane až na pozici prvního prvku v neroztříděném podpoli a přechází do seřazeného podpole.

Uvedená metoda má více algoritmických variant[1]:

Metoda přirozeného bublinového výběru: třídicí cyklus je ukončen tehdy, nedojde-li ve vnitřním cyklu k výměně žádné dvojice.

Ripple-Sort: vylepšuje průběh třídění tím, že si pamatuje pozici, kde došlo k první výměně dvojice. Při hledání dalšího minima začíná vyhledávat až od této pozice.

Shaker-Sort: tato varianta prochází neutříděné podpole zleva-doprava a zprava-doleva. Hledá jak minimum, tak maximum a řadí je na příslušné konce neutříděného podpole, a pro maxima vytváří další třetí utříděné podpole. Algoritmus končí spojením utříděných podpolí minim a maxim, když je neroztříděné podpole prázdné. Při hledání minima i maxima lze využít principu Ripple-Sortu.

Shuttle-Sort: když poprvé dojde k výměně dvou prvků, posunuje metoda minimální prvek doleva tak dlouho, dokud opět nedojde k výměně nebo se minimum dostane na konec řady. Pak se metoda vrátí zpět k první výměně a s maximem postupuje opačným směrem při stejných podmínkách. Metoda končí, když se úspěšně roztřídí poslední dvojice prvků.

bublinové třídění									
zdrojova posloupnost	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9
15	3	3	3	3	3	3	3	3	3
26	15	7	7	7	7	7	7	7	7
7	26	15	15	15	15	15	15	15	15
3	7	26	16	16	16	16	16	16	16
30	16	16	26	19	19	19	19	19	19
25	30	19	19	26	21	21	21	21	21
21	25	30	21	21	26	25	25	25	25
19	21	25	30	25	25	26	26	26	26
16	19	21	25	30	30	30	30	30	30

Obr 3. Schéma bublinového třídění

```
class Bubblesort : TridiciAlgoritmus
    {
        public int[] bubble(int[] mojepole)
        {
            for (int i = 1; i < mojepole.Length; i++) // vnější cyklus třídění
            {
                for (int j = mojepole.Length-1; j >= i; j--)
                {                                     // vnitřní cyklus probublávání
                    if (mojepole[j - 1] > mojepole[j])// porovnání prvků
                    {
                        int pom = mojepole[j -1];     // výměna prvků
                        mojepole[j - 1] = mojepole[j];
                        mojepole[j] = pom;
                        this.pocetvymen = this.pocetvymen + 1;
                    }
                    this.pocetporovnavi = this.pocetporovnavi + 1;
                }
            }
            return mojepole;
        }
```

Program 3 algoritmus bublinového třídění

Metoda bublinového výběru a její variace třídí a porovnává vždy pouze sousední prvky, proto se řadí k méně efektivním metodám s náročností n^2. Porovnáváním sousedních prvků nemůže dojít k změně relativního pořadí stejně velkých prvků, proto je tato metoda stabilní a chová se přirozeně.

Střední hodnota počtu porovnání pro metodu přirozeného bublinového výběru je [2]:

$$C \cong \frac{1}{2} \cdot (n^2 - n)$$

Tato hodnota se používáním různých variant bublinového třídění může snižovat. Jednotlivé varianty eliminují nadbytečné porovnávání prvků. Jejich využívání však neovlivní počty výměn prvků, které jsou výrazně paměťově i časově náročnější [2].

Počet výměn prvků zde závisí na uspořádání prvků před tříděním. V případě setříděného pole se $M_{min} = 0$, a proto se metoda bublinového třídění s výhodou využívá k testování utříděnosti pole. Ve všech ostatních případech je algoritmus bublinového třídění ze všech algoritmů nejméně účinný. Počty výměn pro střední rozložení prvků a opačné upořádaní prvků jsou[1,2]:

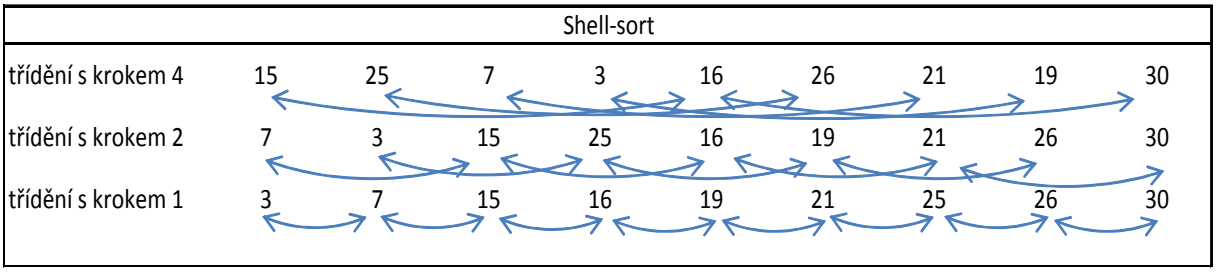
$$M_{střed} \cong \frac{3}{4} \cdot (n^2 - n)$$

$$M_{max} \cong \frac{3}{2} \cdot (n^2 - n)$$

3.4 Třídění vkládáním se zmenšováním kroku

Metoda vylepšuje třídění přímým vkládáním a byla vynalezena v roce 1959 D. L. Shellem, podle něhož je také zvána Shell-sort. Vylepšení tkví v tom, že utříděné prvky se

nepřemisťují o pouze jedno pole, nýbrž o předem daný počet polí, který se postupně zmenšuje. Toto předrozdělení výrazně snižuje potřebný čas k roztřídění pole, a to za předpokladu, že se každý prvek během třídicího procesu posune *n*/3 pozic. Algoritmus rozdělí prvky tříděného pole na 4 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o krok h_3=4 a každou skupinu roztřídí zvlášť pomocí jednoduchého porovnávání. V další etapě jsou všechny prvky pole opět rozděleny na dvě skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o krok h_2 = 2. V poslední etapě seřadíme celou posloupnost všech prvků s krokem h_1 = 1. Poslední etapa dokáže utřídit množinu sama o sobě, nicméně rozdělení procesu třídění do více etap, které v sobě zahrnují jen málo nových změn uspořádání tříděné množiny, je z hlediska efektivnosti algoritmu lepší. Je zřejmé, že každá následující etapa programu těží z výměn prvků v etapě předcházející.



Obr 4 schéma Shellova třídění

```
class Shellsort : TridiciAlgoritmus
    {
        public int[] shell(int[] mojepole)
        {
            int t = 4;                                  // nastavení velikosti prvního
                                                        kroku
            for (int m = -t; m < 0; m++)                // cyklus snižování kroků
            {
                if (m == -3)                            // vynechání kroku o velikosti 3
                { }
                else
                {
                    for (int i = 0; i < mojepole.Length + m; i++)
                    {                                   // vnější cyklus třídění
                        if (mojepole[i] > mojepole[i - m])
                        {                               // porovnání prvků
                            int pom = mojepole[i - m]; // výměna prvků
                            mojepole[i - m] = mojepole[i];
                            mojepole[i] = pom;
                            this.pocetvymen = this.pocetvymen + 1;
                            int zarazka = i + m;        // určení polohy zarážky
                            while (zarazka >= 0)        // vnitřní cyklus třídění
                            {
                                int pam = i;            // uložení původní polohy
                                if (pom < mojepole[i + m])
                                {                       // ověření správné polohy
                                                        tříděného prvku
                                    mojepole[i] = mojepole[i + m];
                                    mojepole[i + m] = pom;
                                                        // výměna prvků
                                    zarazka = zarazka + m;
                                    i = i + m;
                                    this.pocetvymen = this.pocetvymen + 1;
                                }
                                this.pocetporovnavi = this.pocetporovnavi + 1;
```

```
                        zarazka = -1;          // konec vnitřního cyklu
                        i = pam;               // vyvolaní původní polohy
                                               tříděného prvku
                    }
                }
                this.pocetporovnavi = this.pocetporovnavi + 1;
            }
        }
    }
    return mojepole;
}
```

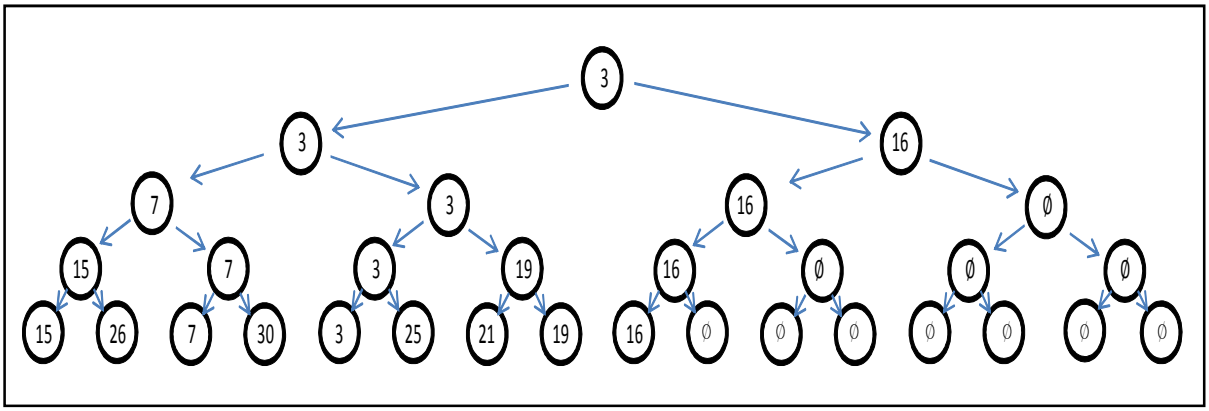
Program 4 algoritmus třídění se zmenšováním kroku

Metoda třídění se zmenšováním kroku již neprovádí operace pouze se sousedními prvky, ale využívá přesunů na větší vzdálenosti. Tato změna oproti předešlým metodám řadí Shell-sort na pomezí účinných algoritmu s časovou a prostorovou náročností $n \cdot log\ n$ a málo efektivních algoritmů s náročností n^2. Shell-sort má časovou a prostorovou náročnost $n^{1,2}$. Metoda není v důsledku výměn na větší vzdálenosti stabilní, nevyužívá pomocné paměti, tj. pracuje in situ. Střední počty porovnání a výměn nebyly odvozeny z důvodu velké matematické složitosti. Další důležitým parametrem při určování efektivnosti Shellova třídění je určení velikosti kroku. Z experimentů vyplývá, že jednotlivé kroky by neměly být vzájemnými násobky, a měla by to být taková čísla, aby docházelo k co nejčastěji interakci mezi jednotlivými skupinami. Mezi nejčastěji volené posloupnosti kroků patří [1,2]:

1) $h_1 = 1,\ h_{i+1} = 2h_i + 1$
2) $h_1 = 1,\ h_2 = 3,\ h_{i+1} = 2h_i - 1$ pro $i > 2$
3) $h_1 = 1,\ h_{i+1} = 3h_i + 1$ kde nejvyšším použitým h_i je nejmenší h_i pro, které platí $h_{j+2} \geq n$

3.5 Stromové třídění

Jedná se o nejvyspělejší metodu založenou na principu výběru. Vylepšení tkví ve způsobu vyhledání minima z tříděného pole. Vhodným rozdělením posloupnosti lze počet potřebných porovnání výrazně snížit. Pro nalezení minima z n prvků je potřeba $n - 1$ porovnání, nicméně pro vyhledání minima z dvojic prvků stačí již $n/2$ porovnání, takto lze dále vyhledávat z dalších n-tic s postupným snižováním potřebných porovnání. Tato myšlenka vedla k vytvoření struktury na základě binárního stromu, který splňuje podmínku, že pro každý prvek v uzlu stromu platí, že všechny prvky v uzlech obou podstromů jsou větší nebo rovné než prvek v daném uzlu. Takováto struktura se nazývá hromada (angl. Heap). Z výše uvedené podmínky vyplývá, že na vrcholu hromady (kořen stromu) je minimální prvek z tříděné množiny.



Obr 5 stromová struktura

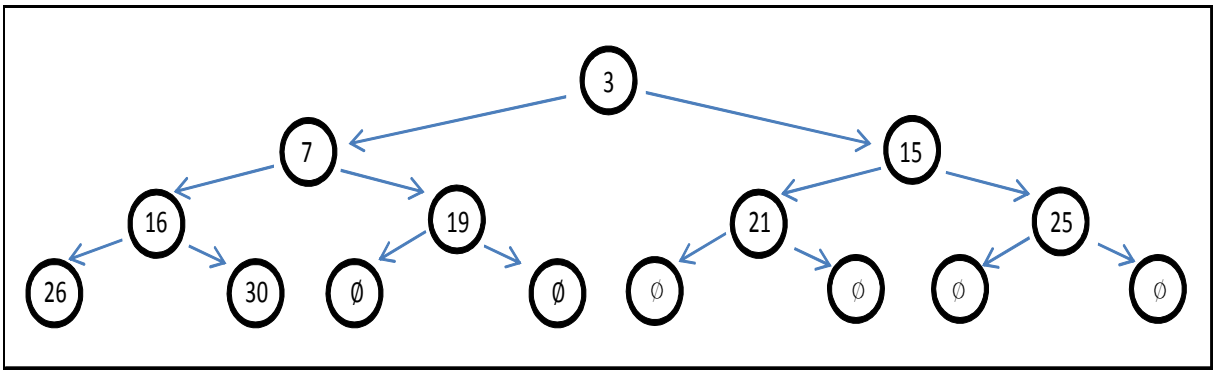
Samotný proces třídění probíhá prosetím celé hromady vybraným prvkem dosazeným na pozici kořene, kdy se tento prvek propadá hromadou na svoje utříděné místo a na jeho místo vyplouvají nižší prvky.

Aby bylo možné správně použít prosetí hromady, je nutné nejdříve hromadu vytvořit. Nejprve nadefinujeme hromadu jako posloupnost prvků h_l, h_{l+1}, …, h_r, takových, že platí

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

pro všechny $i = l \ldots r/2$. Takováto definice svazuje uzly z vyššího patra stromu (otce) s uzly z nižšího patra stromu (pravý a levý syn), z indexu každého z otcovských uzlů lze matematicky vyjádřit indexy všech jemu podřízených uzlů. Dále platí, že index $i = 1$ patří vždy kořenu stromu a odeberou-li se všechny uzly na nejnižší úrovni stromu, bude strom absolutně vyvážený a symetrický. Na nejnižší úrovni stromu l může být maximálně 2^l listů (uzly bez potomků) napojených vždy zleva doprava.

Tvoření hromady probíhá postupným přidáváním prvků z neutříděné množiny do hromady a jejich prosetím na předpokládané místo ve stromu podle výše uvedených pravidel. Konečné roztřídění prvků se provede tak, že se z vytvořené hromady vybere poslední prvek a přiřadí se na místo kořene, ten se naopak přiřadí na místo posledního prvku.



Obr 6 roztříděná hromada

```
class Heapsort : TridiciAlgoritmus
    {
        public int[] heap(int[] mojepole)
        {
```

```
        int a = (mojepole.Length / 2);              // výběr prvního kořene
        int b = mojepole.Length - 1;                // velikost tříděného pole
        for (int L = a; L >= 0; L--)                // cyklus pro vytvoření hromady
        {
            prosit(L, b, mojepole);                 // volání procedury prosetí
        }
        for (int R = b; R >= 1; R--)                // cyklus třídění hromady
        {
            int vymena = mojepole[0];               // výměna prvků
            mojepole[0] = mojepole[R];
            mojepole[R] = vymena;
            pocetvymen = pocetvymen + 1;
            prosit(0, R - 1, mojepole);             // volání procedury prosetí
        }
        return mojepole;
    }
    public void prosit(int l, int r, int[] pole)
    {
        bool jeste = false;                         // podmínka pokračování
        int synové;
        int pom;
        while ((l * 2 <= r) && (!jeste))            // cyklus tvoření stromové
                                                    struktury
        {
            if (l * 2 == r)                         // výběr syna
                synové = l * 2;
            else if (pole[l * 2] > pole[l * 2 + 1])
                                                    // porovnání synů
                synové = l * 2;
            else
                synové = l * 2 + 1;
            this.pocetporovnavi = this.pocetporovnavi + 1;
            if (pole[l] < pole[synové])             // porovnání otce a syna
            {
                pom = pole[l];                      // výměna prvků
                pole[l] = pole[synové];
                pole[synové] = pom;
                this.pocetvymen = this.pocetvymen + 1;
                l = synové;                         // přechod na nižší úroveň
                                                    stromu
            }
            else
            {
                jeste = true;                       // ukončení prosetí
            }
            this.pocetporovnavi = this.pocetporovnavi + 1;
        }
    }
```

Program 5 algoritmus stromového třídění

Metoda třídění hromadou postupně přesouvá velké prvky při tvoření hromady nejdříve na levou stranu od kořene a při samotném roztřídění je umístí na jejich právoplatné místo, proto se nedoporučuje využívat pro malé pole s nízkým počtem prvků *n*. S rostoucím počtem prvků v poli roste i účinnost této metody. Celková časová a prostorová náročnost algoritmu je $n \cdot \log n$. Z důvodu přesunů na velké vzdálenosti je metoda nestabilní, nechová se přirozeně a nevyužívá pomocné paměti tj. pracuje in situ[1].

Na vytvoření hromady je nutné vyvolat proceduru prosetí maximálně $n/2$ *krát*, během kterých se prvky přesunou přes $\log n/2$, $\log n/2-1$ … $\log n-1$ pozic logaritmus má základ 2. Na samotné třídění algoritmus vyvolá proceduru prosetí $n-1$ krát, přičemž přesune prvky $\log n-1$,

log *n*-2, …, 1. Nakonec ještě spotřebuje *n*-1 přesunů na převedení prvku z kořene stromu na pravou část hromady. Celkový počet přesunů prvku pro třídění hromadou je přibližně:

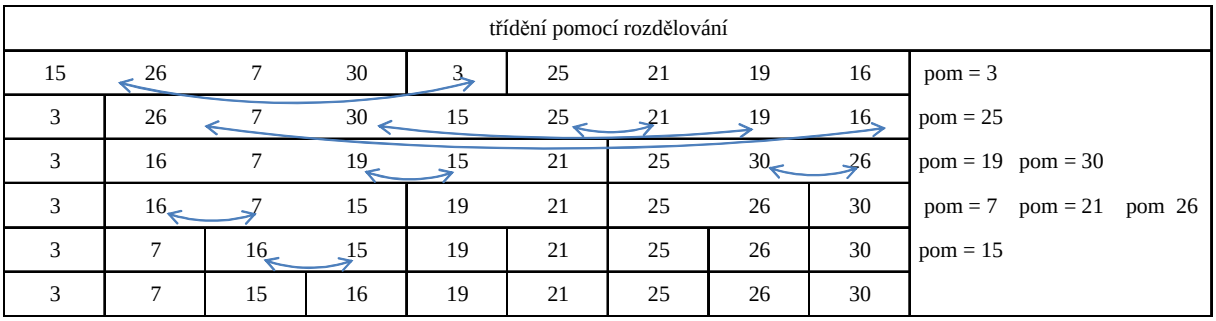
$$M_{střed} \cong \frac{1}{2} n \cdot \log n$$

Zatím nebylo určeno, v jakých případech má tato metoda nejlepší nebo nejhorší výsledky. Všeobecně se jeví jako nejvhodnější pole pro třídění hromadou takové, které je přibližně opačně uspořádané. A to díky tomu, že fáze tvoření hromady v tomto případě nepotřebuje žádné přesuny[2].

3.6 Třídění na principu rozdělování

Třídění rozdělování je jednou z nejefektivnějších metod třídění polí. Vychází z metody využívající princip výměny prvků. Autorem této metody je C. A. Hoare, který ji pojmenoval Quicksort (překlad rychlé třídění). Algoritmus využívá zásady „rozděluj a panuj“ a to tak, že rozdělované pole *P* rozdělí na podpole takové, že pro ně platí $P_1 < P_2 \ldots < P_k$. Toto rozdělení se stále opakuje pro všechna vzniklá podpole, dokud má podpole alespoň dva prvky (jejich rozdělením se stává pole utříděné). Využívá se premisy, že nejefektivnější výměny jsou ty na velké vzdálenosti.

Rozdělování na podpole probíhá tak, že se do pomocné proměnné *pom* přiřadí libovolný prvek z neutříděného pole a začne se porovnávat nejprve s prvky z levého konce pole, dokud platí relace $a_i < pom$. Poté se celý proces opakuje pro prvky z pravého konce pole za použití relace $a_j > pom$. Pokud se najdou takové dva prvky a_i a a_j, které nesplňují tyto relace, vzájemně si vymění pozice. Celý proces se opakuje, dokud se prohledávání nesetká uprostřed pole.



Obr 6 schéma třídění pomocí rozdělování

Výsledek tohoto procesu je rozdělení pole na tři části, z nichž první obsahuje pouze prvky menší než zvolená proměnná *pom* a druhá obsahuje pouze prvky větší než zvolená proměnná *pom* a ve třetí části je pomocná proměnná *pom*. Celý postup se opakuje, dokud se nerozdělí všechna podpole obsahující pouze dvojice prvků, a tím dojde k utřídění pole.

Na účinnost této metody má velmi významný vliv hodnota pomocné proměnné *pom*, podle které se rozděluje tříděné pole. V nejhorším případě se může do pomocné proměnné *pom* dosadit nejmenší nebo největší prvek z tříděné množiny, což má za následek nerovnoměrné rozdělení na dvě podmnožiny, přičemž jedna obsahuje pouze jeden prvek a ve druhé jsou všechny ostatní prvky. V nejlepším případě se pole má rozdělit na dva stejně velké úseky. Aby se podařilo získat rovnoměrné rozdělení, je důležité za hodnotu pomocné proměnné *pom* zvolit ten nejvhodnější prvek, a tím je medián. Medián je takové číslo, pro které platí, že právě polovina prvků v souboru je menší než toto číslo a druhá polovina je

větší. Nejlépe se medián určuje jako prostřední hodnota z utříděného pole. Další možnosti určení mediánu značně zvyšují časové nároky na algoritmus, proto se za pomocnou proměnou *pom* volí prostřední prvek z tříděného pole.

```
class Quicksort : TridiciAlgoritmus
    {
        public int[] quick(int[] mojepole)
        {
            int l = 0;                                  // nastavení počátku rozdělované
                                                        části
            int r = mojepole.Length - 1;                // nastavení konce rozdělované
                                                        části
            rozdel(l, r, mojepole);                     // volání rozdělovací procedury
            return mojepole;
        }
        public void rozdel(int a, int b, int[] pole)
        {
            int i = a;                                  // nastavení počátku rozdělované
                                                        části
            int j = b;                                  // nastavení konce rozdělované
                                                        části
            int pom = pole[(a + b) / 2];                // výběr prvku pro rozdělení
                                                        pole
            do
            {
                while (pole[i] < pom)                   // vyhledávání většího prvku
                {
                    i = i + 1;
                    this.pocetporovnavi = this.pocetporovnavi + 1;
                }
                while (pom < pole[j])                   // vyhledávání menšího prvku
                {
                    j = j - 1;
                    this.pocetporovnavi = this.pocetporovnavi + 1;
                }
                if (i <= j)                             // kontrola polohy prvků
                {
                    int vymena = pole[i];               // výměna prvků
                    pole[i] = pole[j];
                    pole[j] = vymena;
                    this.pocetvymen = this.pocetvymen + 1;
                    i = i + 1;                          // pokračování vyhledávání
                                                        většího prvku
                    j = j - 1;                          // pokračování vyhledávání
                                                        menšího prvku
                }
            } while (i < j);                            // konec cyklu rozdělování
            if (a < j)                                  // rekurzivní volání procedury
                                                        rozděl pro první část pole
            {
                rozdel(a, j, pole);
            }
            if (i < b)                                  // rekurzivní volání procedury
                                                        rozděl pro druhou část pole
            {
                rozdel(i, b, pole);
            }
        }
```

Program 6 algoritmus třídění na principu rozdělování

V průběhu rozdělování do částí podle zvoleného prvku se ostatní prvky přesouvají na velmi velké vzdálenosti, což vede k vysoké efektivitě tohoto algoritmu. Těmito přesuny se však metoda stává nestabilní, protože porušuje relativní pořadí stejně velkých prvků před a po třídění. Navíc se nechová přirozeně, protože setřídění náhodně uspořádaného pole prvků trvá mnohem déle než setřídění uspořádaného nebo opačně uspořádaného pole. Využitím rekurze vzniká potřeba dodatečného využití pomocné paměti, metoda tudíž nepracuje in situ. Rekurze také zvyšuje potřebný čas na roztřídění pole. Ale ani všechny tyto nepříznivé vlastnosti neovlivňují Quick-sort natolik, aby nešlo říci, že je to jedna z nejlepších metod vnitřního třídění polí, které existují. Časová náročnost tohoto algoritmu je $n \cdot log\ n$[1,2].

Pro přeskupení pole podle zvoleného prvku je potřeba n porovnání, kde n je počet prvků v přeskupované části pole. Pro náhodně vybraný prvek z náhodného uspořádaného pole platí, že počet výměn prvků při rozdělování je[2]:

$$M_{střed} \cong \frac{n}{6} - \frac{1}{6n}$$

V optimálním případě, kdy za pomocnou proměnou *pom*, podle které se pole rozděluje, zvolíme medián, bude potřeba k roztřídění pole vyvolat proceduru *rozděl* přesně log n krát, celkový počet porovnání a výměn prvků bude[2]:

$$C_{střed} = n \cdot \log n$$

$$M_{střed} = \frac{n}{6} \cdot \log n$$

Pravděpodobnost, že vždy vybereme medián, je velmi malá. Náhodně zvoleným prvkem ze středu rozdělované části se však výše uvedené hodnoty počtů porovnání a výměn prvků změní jen o faktor $2 \cdot \ln 2$[2].

Nejhorší případ nastává, když se za hodnotu pomocné proměnné *pom*, podle které se rozděluje pole, vybere krajní prvek, tedy největší nebo nejmenší prvek pole. Následuje situace, kdy se pole rozdělí na první část obsahující jeden prvek a druhou, která obsahuje všechny ostatní. Pokud by se takové schéma opakovalo, může se časová a prostorová náročnost zvýšit až na hodnotu n^2. Doporučenou metodou, jak se tomuto scénáři vyhnout, je dosadit do pomocné proměné *pom* jako medián malého vzorku prvků z rozdělovaného podpole[2].

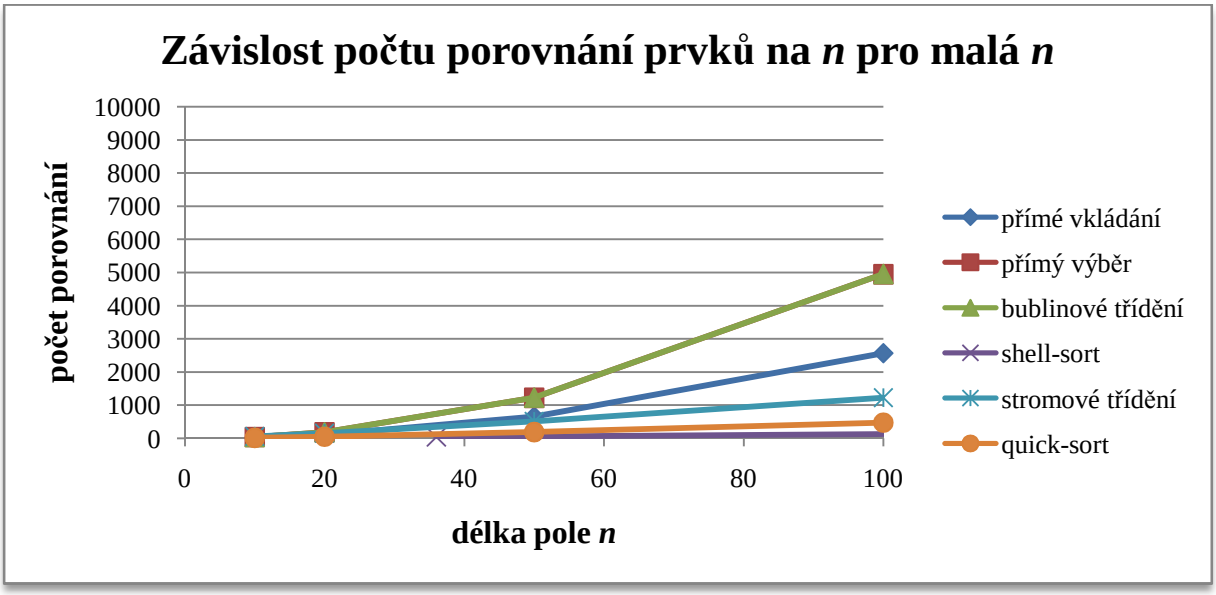
4 Experimentální ověření časové náročnosti

Výše uvedené algoritmy byly zpracovány do experimentálního programu, který při procesu třídění počítá pro každou metodu zvlášť potřebné počty porovnání a výměn prvků pro zpracování daného pole. *Bakalarkaprogram* byl vytvořen v programovacím jazyce C# v prostředí Microsoft Visual C# 2010 Express. Experimenty byly provedeny na počítači Toshiba SATELLITE A300 – 1EG, procesor Intel Core 2 Duo T5750 (2 MB L2 cache, 2 GHz, 667 MHz FSB), 3072 MB DDR2 RAM (667 MHz), systém Microsoft Windows Vista Home Premium.

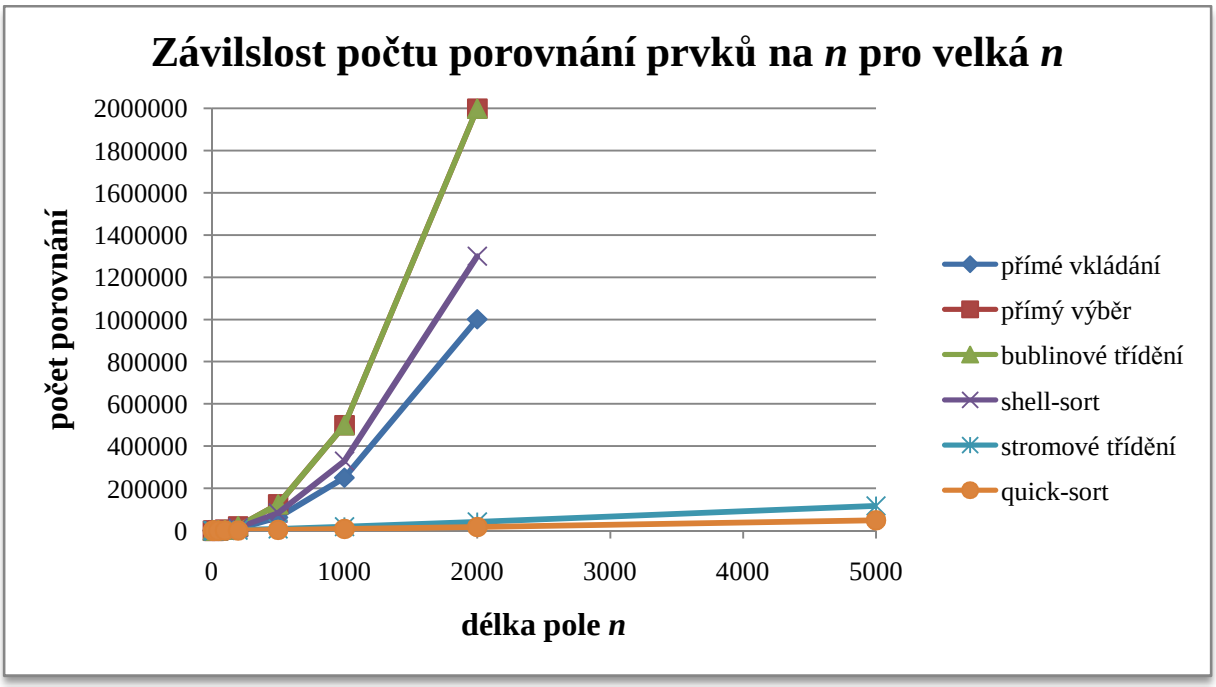
Jednotlivé algoritmy třídění byly zpracovány do vlastních objektů, dále program obsahuje ještě objekt *Pole*, *TridiciAlgoritmus* a *Program*. V objektu *TridiciAlgoritmus* se inicializují společné vlastnosti a metody všech třídicích algoritmů, mezi něž patří metoda pro výpis pole, a hlavně se zde inicializují proměnné *pocetvymen* a *pocetporovnani*, které slouží jako indikátory časové náročnosti jednotlivých algoritmů. V objektu *Pole* probíhá generace náhodných polí o velikosti, kterou si volí uživatel. Generování pole využívá systémového generátoru náhodných čísel *Random,* jednotlivé prvky se volí z rozmezí čísel 1 až 9999. Objekt *Program* obsahuje uživatelské rozhraní a konstruktory jednotlivých objektů. Uživatel si zvolí délku pole pro porovnání, následuje cyklus o 1000 průchodech, který při každém průchodu vygeneruje pole o zadané délce. Tato pole jsou postupně roztříděna všemi algoritmy, které přitom počítají počty potřebných porovnání a výměn. V posledním průchodu cyklu se navíc vypíše vygenerované neutříděné pole, a pro každou metodu utříděné pole a aritmetický průměr počtu porovnání a výměn pro 1000 vygenerovaných polí. Celková časová náročnost je hodnocena jako součet počtu porovnání a počtu výměn.

tabulka počtu porovnání v závislosti na počtu prvků v poli *n*										
délka pole		10	20	50	100	200	500	1000	2000	5000
počet porovnání	přímé vkládání	29	111	658	2572	10099	62855	251002	1001516	1958984
	přímý výběr	45	190	1225	4951	19900	124750	499500	1999000	
	bublinové třídění	45	190	1225	4950	19900	124750	499500	1999000	
	shell-sort	36	139	865	3459	13708	83416	329316	1299520	
	stromové třídění	59	155	517	1231	2857	8432	18859	41712	117691
	quick-sort	18	54	192	476	1145	3450	7832	17326	48948

Tab. 2 počet porovnání prvků v závislosti na velikosti pole n



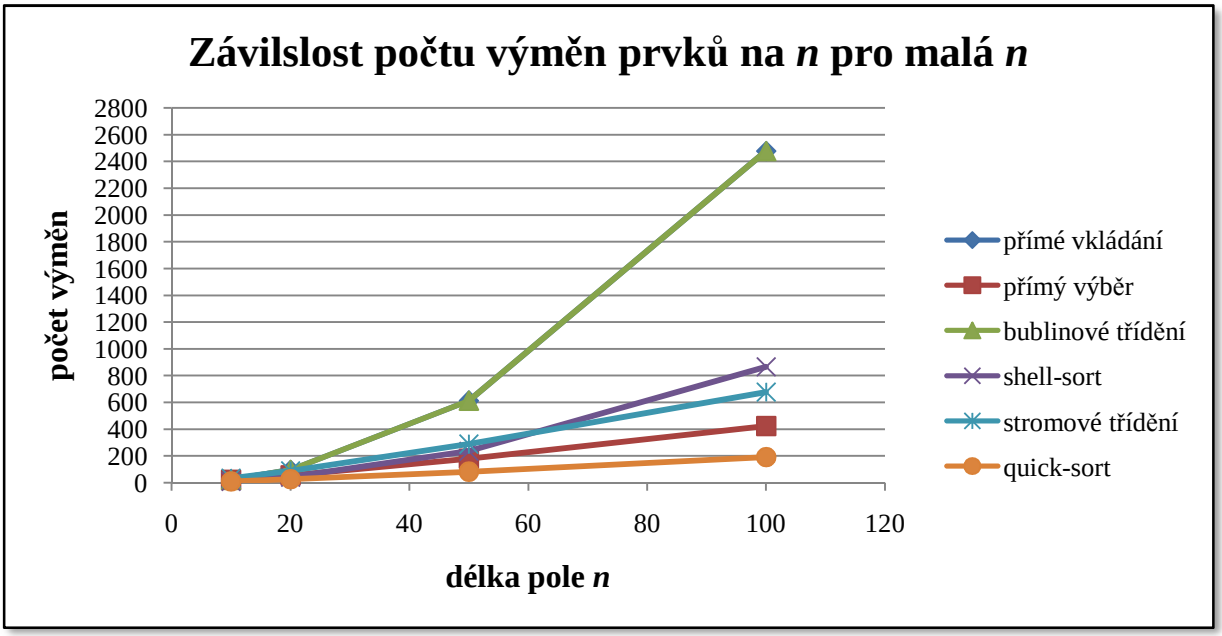
Graf 1 závislost počtu porovnání prvků na velikosti pole n pro malá n



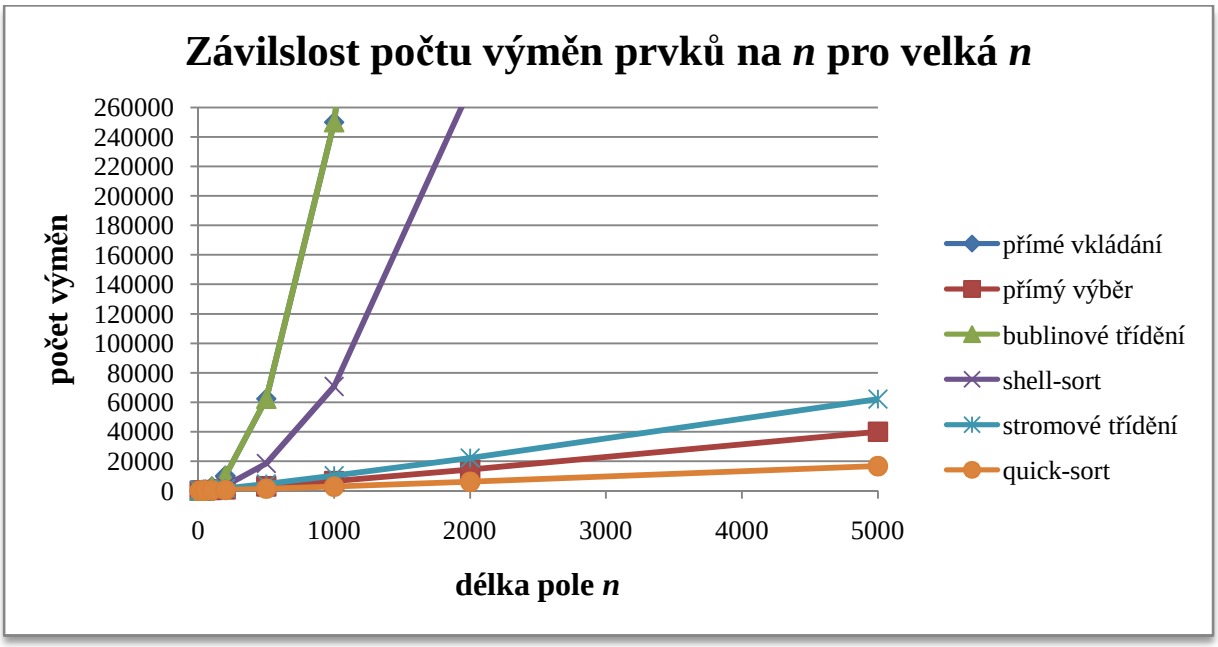
Graf 2 závislost počtu porovnání prvků na velikosti pole n pro velká n

tabulka počtu výměn v závislosti na počtu prvků v poli *n*										
délka pole		10	20	50	100	200	500	1000	2000	5000
počet výměn	přímé vkládání	22	95	613	2477	9904	62362	250010	999524	1953993
	přímý výběr	22	55	179	424	980	2897	6472	14258	39895
	bublinové třídění	22	95	613	2477	9958	62362	250010	999524	1953993
	shell-sort	12	43	236	867	3216	18435	70724	273117	1654334
	stromové třídění	34	88	289	678	1557	4540	10078	22155	62099
	quick-sort	11	27	84	193	433	1240	2721	5938	16603

Tab. 3 počet výměn prvků v závislosti na velikosti pole n



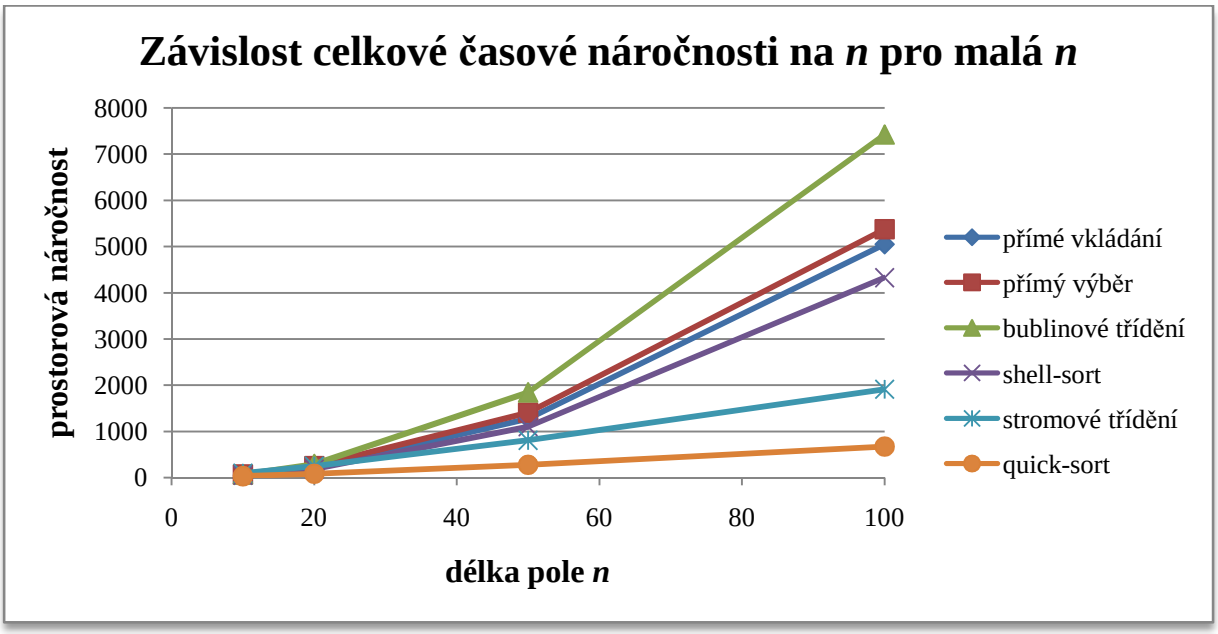
Graf 3 závislost počtu výměn prvků na velikosti pole n pro malá n



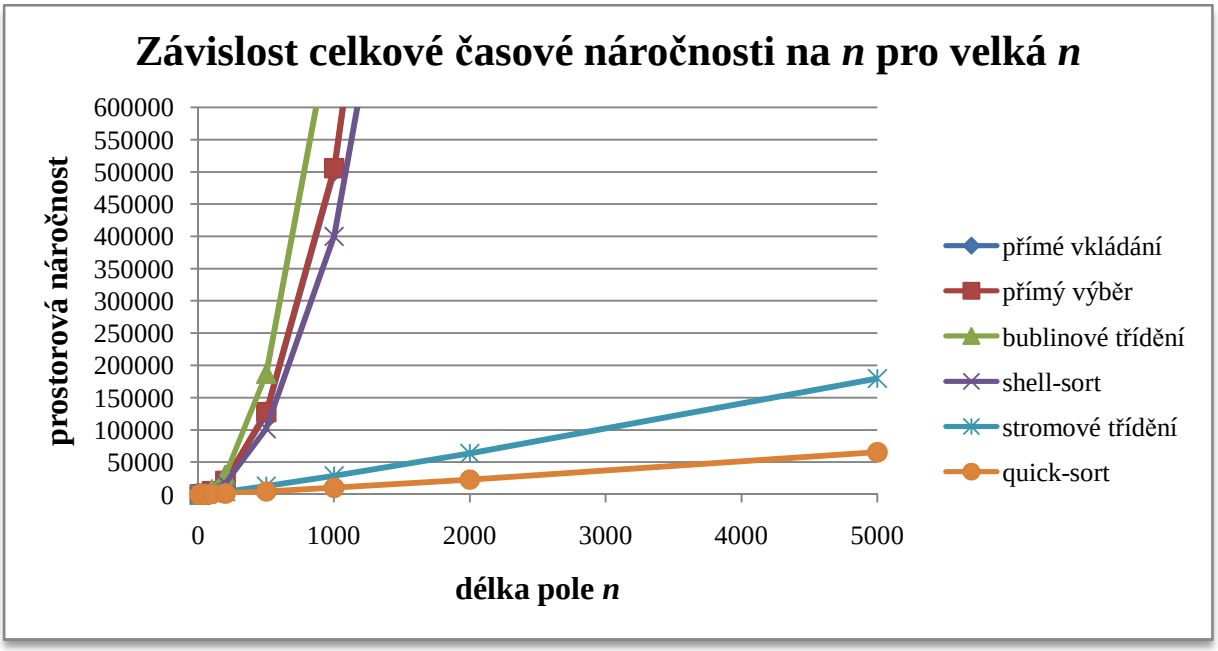
Graf 4 závislost počtu výměn prvků na velikosti pole n pro velká n

tabulka celkové časové náročnosti v závislosti na počtu prvků v poli *n*										
délka pole		10	20	50	100	200	500	1000	2000	5000
prostorová náročnost	přímé vkládání	51	206	1271	5049	20003	125217	501012	2001040	3912977
	přímý výběr	67	245	1404	5375	20880	127647	505972	2013258	
	bublinové třídění	67	285	1838	7427	29858	187112	749510	2998524	
	shell-sort	48	182	1101	4326	16924	101851	400040	1572637	
	stromové třídění	93	243	806	1909	4414	12972	28937	63867	179790
	quick-sort	29	81	276	669	1578	4690	10553	23264	65551

Tab. 4 celková časová náročnost v závislosti na počtu prvků v poli n



Graf 5 závislost celkové časové náročnosti na velikosti pole n pro malá n



Graf 6 závislost celkové časové náročnosti na velikosti pole n pro velká n

5 Závěr

V úvodní části bakalářské práce bylo provedeno shrnutí základních pojmů v oblasti algoritmů třídění, rozdělení metod podle různých parametrů, byla vymezena hlavní kriteria pro porovnávání algoritmů a naznačeny předpokládané výsledky pro různě složité algoritmy.

Následoval popis vybraných metod třídění s názorným rozborem mechanismu třídění, zápis zdrojového kódu v jazyce C# a analýza efektivnosti programů.

Z experimentálních výsledků vyplývá, že první čtyři algoritmy jsou výrazně náročnější než zbylé, což potvrzuje očekávaný výsledek. Bublinové třídění se ukázalo jako nejméně efektivní způsob vnitřního třídění polí. Metoda přímého výběru a metoda přímého vkládání se ukázaly jako rovnocenní soupeři i přes fakt, že metoda přímého výběru potřebuje výrazně méně výměn prvků. Shellovo třídění vykazuje o trošku lepší výsledky než předchozí tři metody, nicméně i tak je velmi náročná na počty porovnání a výměn. Mezi efektivní metody třídění polí se svými výsledky zařadilo stromové třídění a třídění pomocí rozdělování, které je ještě asi dvaapůlkrát efektivnější. Tak dobrých výsledků dosahuje díky přesunům na velmi velké vzdálenosti napříč tříděným polem.

Použitá literatura

[1] HONZÍK, J. a kol. Programovací techniky. Skripta. VUT v Brně, 1985
[2] WIRTH, N. Algoritmy a štruktúry údajov. Bratislava, ALFA 1988.
[3] www.publicjoe.f9.co.uk/csharp/sort02.html
[4] www.microsoft.com/express/
[5] www.gnu.org/software/octave/
[6] www.fme.vutbr.cz
[7] www.autnt.fme.vutbr.cz

Seznam příloh