Třídící algoritmy a jejich využití

Sorting algorithms and their utilization

Bc. Pavel Mlček

Diplomová práce
2008

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2007/2008

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Pavel MLČEK**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Třídící algoritmy a jejich využití**

Zásady pro vypracování:

1. **Vytvořte literární rešerši na zadané téma.**
2. **Sezamte se s nejčastěji používanými třídícími algoritmy.**
3. **K těmto algoritmům proveďte podrobnou analýzu a na základě získaných informací je porovnejte z hlediska rychlosti, kvality a využití.**
4. **Vytvořte programovou knihovnu třídících algoritmů založenou na objektovém programování. K této knihovně posléze vytvořte funční aplikaci. V této aplikaci otestujte implementované třídicí algoritmy a výsledky porovnejte s informacemi získanými z jiných zdrojů.**
5. **K tvorbě knihovny i celé aplikace použijte programovací jazyk C/C++ a knihovnu WX Widgets.**
6. **K vytvořené knihovně vytvořte dokumentaci, která bude popisovat způsob implementace do jiných programů.**

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **BLIŽňÁK, Michal. Systémové programování. Zlín : Univerzita Tomáše Bati, 2005. 202 s. ISBN 80-7318-364-1.**
2. **WROBLEWSKI, Piotr. Algoritmy - datové struktury a programovací techniky. 1. vyd. Brno : Computer Press, 2004. 351 s. ISBN 80-251-0343-9.**
3. **LIBERTY, Jesse. Naučte se C++ za 21 dní. Brno : Computer Press, 2007. 796 s. ISBN 978-80-251-1583-1.**
4. **PETZOLD, Charles. Programování ve Windows. 1. vyd. Praha : Computer Press, 1999. 1216 s. ISBN 80-7226-206-8.**
5. **SCHILDT, Herbert. Nauč se sám C. Praha : SoftPress, 2001. 624 s. ISBN 80-86497-16-X.**
6. **SCHILDT, Herbert. Nauč se sám C++. Praha : SoftPress, 2001. 624 s. ISBN 80-86497-13-6.**
7. **WxWidgets [online]. 2002 [cit. 2008-01-23]. Dostupný z WWW: http://www.wxwidgets.org/.**

Vedoucí diplomové práce: **Ing. Pavel Pokorný, Ph.D.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2008**

Termín odevzdání diplomové práce: **19. května 2008**

Ve Zlíně dne 20. února 2008

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Tato diplomová práce se zabývá řadícími algoritmy, jejich popisem, porovnáním a implementací.

V první části práce popisuji široké téma třídění a řadících algoritmů. Charakterizuji jejich principy, vlastnosti a provádím jejich analýzu.

Ve druhé části mé práce se zabývám implementací řadících algoritmů v jazyce C++ a popisuji tvorbu aplikace použité na testování řadicích algoritmů.. Na závěr této části algoritmy testuji a vyhodnocuji výsledky.

Klíčová slova: Algoritmus, složitost algoritmu, třídění, řadící algoritmy, C++, wxWidgets

ABSTRACT

This diploma thesis deals with sorting algorithms, their description, comparison and implementation.

In the first part of the thesis, I describe the wide theme of sorting and sorting algorithms. I characterize their principles, properties and analyze them.

In the second part, I deal with implementation of sorting algorithms in C++ language and describe a creation of the application used for testing of sorting algorithms. At the end of this part, I test the algorithms and evaluate the results.

Keywords: Algorithm, complexity of algorithm, sorting, sorting algorithms, C++, wxWidgets

Na tomto místě bych rád poděkoval vedoucímu mé diplomové práce panu Ing.Pavlu Pokornému, Ph.D. za odborné vedení, cenné rady a připomínky, které jsem využil při realizaci této práce. Dále bych rád poděkoval své rodině a přátelům za trpělivost.

Prohlašuji, že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků, je-li to uvolněno na základě licenční smlouvy, budu uveden jako spoluautor.

Ve Zlíně …….

Podpis diplomanta

OBSAH

ÚVOD **10**

I TEORETICKÁ ČÁST **11**

1 ALGORITMY **12**

1.1 ALGORITMUS A JEHO DEFINICE 12

1.2 VLASTNOSTI ALGORITMŮ 13

1.2.1 Jednoznačnost 13
1.2.2 Opakovatelnost 13
1.2.3 Hromadnost 13
1.2.4 Rezultativnost 13
1.2.5 Efektivnost 13
1.2.6 Numerická stabilita 13

1.3 DRUHY ALGORITMŮ 14

2 ANALÝZA VÝKONNOSTI ALGORITMŮ **15**

2.1 SLOŽITOST ALGORITMU 15

2.1.1 Časová složitost 15
2.1.2 Paměťová složitost 16

2.2 ODHADY SLOŽITOSTI ALGORITMŮ 17

2.2.1 Horní odhad složitosti algoritmu 17
2.2.2 Dolní odhad složitosti algoritmu 18
2.2.3 Očekávaná složitost algoritmu 18

2.3 PRAKTICKÉ POROVNÁVÁNÍ ALGORITMŮ 19

3 TŘÍDĚNÍ **20**

3.1 DEFINICE PROBLÉMU 20

3.2 VYSVĚTLENÍ POJMŮ 20

4 ŘADÍCÍ ALGORITMY **22**

4.1 KLASIFIKACE ŘADÍCÍCH ALGORITMŮ 22

4.1.1 Podle typu paměti v níž je řazená struktura uložena 22
4.1.2 Podle typu procesoru 22
4.1.3 Podle stability 23
4.1.4 Podle velikosti dodatečné pracovní paměti 23
4.1.5 Podle chování na částečně seřazených souborech dat 23
4.1.6 Podle způsobu využití klíčů 23
4.1.7 Podle základního principu řazení 24

5 ŘAZENÍ NA PRINCIPU VÝBĚRU **25**

5.1 SELECTION SORT 25

5.1.1 Popis algoritmu 25
5.1.2 Analýza algoritmu 25

5.2 HEAP SORT 26

5.2.1 Halda 26
5.2.2 Popis algoritmu 27

5.2.3 Analýza algoritmu 27
5.2.4 Varianty algoritmu Heap Sort 28
5.3 INTRO SORT 29
5.4 TERNARY HEAP SORT 29
5.5 SMOOTH SORT 29
5.6 BUBBLE SORT 29
5.6.1 Popis algoritmu 29
5.6.2 Analýza algoritmu 30
5.6.3 Varianty algoritmu Bubble Sort 30
5.7 RIPPLE SORT 30
5.8 SHAKER SORT 30
5.9 SHUTTLE SORT 31
5.10 GNOME SORT 31
5.10.1 Popis algoritmu 31
5.10.2 Analýza algoritmu 31
6 ŘAZENÍ NA PRINCIPU VKLÁDÁNÍ 32
6.1 INSERTION SORT 32
6.1.1 Popis algoritmu 32
6.1.2 Analýza algoritmu 32
6.1.3 Varianty algoritmu Insertion Sort 33
6.2 BINARY INSERTION SORT 33
6.3 BINARY TREE SORT 33
6.4 LIBRARY SORT 33
6.4.1 Analýza algoritmu 34
6.5 SHELL SORT 34
6.5.1 Popis algoritmu 35
6.5.2 Analýza algoritmu 35
7 ŘAZENÍ NA PRINCIPU ROZDĚLENÍ 37
7.1 QUICK SORT 37
7.1.1 Popis algoritmu 37
7.1.2 Volba pivota 37
7.1.3 Analýza algoritmu 38
7.1.4 Varianty algoritmu Quick Sort 39
7.2 NON-RECURSIVE QUICK SORT 39
8 ŘAZENÍ NA PRINCIPU SETŘÍDĚNÍ 40
8.1 MERGE SORT 40
8.1.1 Popis algoritmu 40
8.1.2 Analýza algoritmu 40
9 ŘAZENÍ NA JINÝCH PRINCIPECH 42
9.1 BUCKET SORT 42
9.1.1 Popis algoritmu 42

9.1.2 Analýza algoritmu 42
9.2 COUNTING SORT 43
9.2.1 Analýza algoritmu 43
9.3 RADIX SORT 43
9.3.1 Popis algoritmu LSD Radix Sort 44
9.3.2 Popis algoritmu MSD Radix Sort (rekurzivního) 44
9.3.3 Analýza algoritmu 44
9.4 BOGO SORT 45
9.4.1 Popis algoritmu: 45
9.4.2 Analýza algoritmu 45
10 PRINCIPY VNĚJŠÍCH SEKVENČNÍCH ŘADÍCÍCH METOD 46
10.1 METODA PŘÍMÉHO SETŘIĎOVÁNÍ (STRAIGHT MERGING) 46
10.2 METODA PŘÍMÉHO VYVÁŽENÉHO SETŘIĎOVÁNÍ (STRAIGHT BALANCED MERGING) 46
10.3 METODA PŘIROZENÉHO SETŘIĎOVÁNÍ (NATURAL MERGING) 46
10.4 METODA PŘIROZENÉHO VYVÁŽENÉHO SETŘIĎOVÁNÍ (NATURAL BALANCED MERGING) 47
10.5 METODA MNOHACESTNÉHO VYVÁŽENÉHO SETŘIĎOVÁNÍ (BALANCED MULTIWAY MERGING) 47
11 JAZYKY C, C++ A KNIHOVNA WXWIDGETS 49
11.1 JAZYK C 49
11.2 JAZYK C++ 49
11.2.1 Historie C++ 49
11.2.2 Objektově orientované programování 50
11.2.3 Zapouzdření 50
11.2.4 Dědičnost 50
11.2.5 Polymorfismus 51
11.3 MULTIPLATFORMNÍ SOTWAROVÁ KNIHOVNA WXWIDGETS 51
11.3.1 Historie wxWidgets 51
II PRAKTICKÁ ČÁST 52
12 KONCEPCE TESTOVÁNÍ ŘADÍCÍCH ALGORITMŮ 53
12.1 TESTOVACÍ PŘÍPADY 53
13 KNIHOVNA ŘADÍCÍCH ALGORITMŮ 55
13.1 VÝVOJOVÉ NÁSTROJE POUŽITÉ KE TVORBĚ KNIHOVNY 55
13.2 ŠABLONY 55
13.3 TŘÍDA MYSTORAGE 55
13.3.1 Atributy třídy myStorage 56
13.3.2 Konstruktor třídy myStorage 57
13.3.3 Řadící metody třídy myStorage 57
13.3.4 Metody třídy myStorage pro výpis prvků 58
13.3.5 Metody třídy myStorage pro generování posloupností prvků 58

14 TESTOVACÍ APLIKACE 60
14.1 VÝVOJOVÉ NÁSTROJE POUŽITÉ KE TVORBĚ TESTOVACÍ APLIKACE 60
14.2 VYTVOŘENÍ KOSTRY APLIKACE 60
14.3 VYTVOŘENÍ GRAFICKÉHO UŽIVATELSKÉHO ROZHRANÍ 60
14.3.1 Rozmístění ovládacích prvků 61
14.4 POPIS IMPLEMENTACE KNIHOVNY MYSTORAGE DO TESTOVACÍ APLIKACE 62
14.5 POPIS APLIKACE 63
15 TESTOVÁNÍ ŘADÍCÍCH ALGORITMŮ 67
15.1 TESTOVACÍ PODMÍNKY 67
15.2 ŘAZENÍ POSLOUPNOSTI 100 PRVKŮ 68
15.3 ŘAZENÍ POSLOUPNOSTI 5.000 PRVKŮ 70
15.4 ŘAZENÍ POSLOUPNOSTI 100.000 PRVKŮ 72
ZÁVĚR 74
SEZNAM POUŽITÉ LITERATURY 75
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 78
SEZNAM OBRÁZKŮ 79
SEZNAM GRAFŮ 80
SEZNAM TABULEK 81
SEZNAM PŘÍLOH 82

ÚVOD

Dá se říci, že pojem třídění je starý jako lidstvo samo. Už v dávných časech měli lidé potřebu své věci třídit a udržovat si tak ve věcech pořádek a řád. Tato potřeba se neustále s lidstvem vyvíjí až do současnosti. Stále máme potřebu třídit si věci na pracovním stole, peníze v peněžence, či knížky na poličce.

S bouřlivým rozvojem počítačové techniky ve 20. století se však objevuje potřeba nová, a tou je třídění dat. Techniky, které se tříděním dat v našich počítačích zabývají, se nazývají třídící, respektive řadící algoritmy. Tyto algoritmy prošly ve dvacátém století stejným „raketovým“ rozvojem, jako celá široká oblast informatiky.

Ač se obecně tvrdí, že řadící algoritmy jsou jedny z nejstudovanějších algoritmů vůbec, tak a to především na české publikační scéně chybí ucelenější a komplexnější publikace, zabývající se pouze tímto tématem. Když se řekne řadící algoritmus, tak každého napadne obligátní bublinkové řazení, ale málokdo si uvědomí, jak pestré a obsáhlé téma se v řadících algoritmech ve skutečnosti nachází.

Cílem mé teoretické části práce bylo seznámit se s co nejširším okruhem řadících algoritmů a pokusit se tento okruh zmapovat a sepsat v podobě literární rešerše. To jsem také učinil a věřím, že má práce v budoucnu pomůže některým studentům, nebo třeba jen zájemcům o tuhle problematiku pochopit základní principy řadících algoritmů a zorientovat se v této rozsáhlé problematice.

V praktické části práce jsem vytvořil a popsal programovou knihovnu nejpoužívanějších řadících algoritmů v programovacím jazyce C++. S pomocí této mnou vytvořené knihovny a multiplatformní knihovny wxWidgets jsem následně vytvořil funkční testovací aplikaci těchto algoritmů a algoritmy podrobil testům, jejichž výsledky jsou umístěny a analyzovány na konci mé praktické části práce.

I. TEORETICKÁ ČÁST

1 ALGORITMY

Pojem algoritmus je cca 1200 let starý a souvisí s významným perským astronomem a matematikem al-Khwarizmim žijícím v 9. století našeho letopočtu. Vznikl ve 12. století zkomolením jeho příjmení „al-Khwarizmi“ na „Algorizmi“ při překladu jeho díla věnovaného základům algebry. Tento termín se v různých obměnách používal pro popis nejrůznějších matematických postupů. Pojem „algoritmus“ se v dnešním slova smyslu používá zhruba od počátku 20. století.

1.1 Algoritmus a jeho definice

Algoritmus je obecný předpis sloužící pro řešení zadaného problému. Představuje posloupnost kroků doplněných jednoznačnými pravidly. Pravidla jsou dána množinou povolených operací, které je možno při řešení aplikovat v různém pořadí. Algoritmus tvoří tři části:

- Vstupní data
- Obecný předpis řešení
- Výstupní data

Algoritmus zabezpečuje, že je množině vstupních dat přiřazena odpovídající množina výstupních dat splňujících zadanou podmínku. Někdy je také nazýván „mlýnkem“ na data. Nasypeme do něj různé informace, otočíme klikou a dostaneme výsledek. Algoritmy se objevují prakticky ve všech vědních oborech, nejčastěji se však používají v technických vědách. Setkáváme se s nimi v širším slova smyslu i v běžném životě ve formě různých návodů a předpisů (např.: návod k použití přístroje, kuchyňský recept, lékařský předpis nebo pravidla silničního provozu). Popis algoritmu může být realizován různými způsoby, od slovního popisu až k zápisu v nějakém formálním jazyce. [1]

1.2 Vlastnosti algoritmů

Každý algoritmus by měl splňovat následující podmínky:

1.2.1 Jednoznačnost

Každý algoritmus je složen z kroků, které na sebe navazují. Každý krok můžeme charakterizovat jako přechod z jednoho stavu algoritmu do jiného, přičemž každý stav je určen zpracovávanými daty. Tím, jak data v jednotlivých stavech vypadají, musí být jednoznačně určeno, který krok následuje.

1.2.2 Opakovatelnost

Při použití stejných vstupních údajů musí algoritmus dospět vždy k témuž výsledku. [2]

1.2.3 Hromadnost

Algoritmus lze použít pro řešení stejné třídy problémů s různými vstupními hodnotami, pro jejichž libovolnou kombinaci obdržíme jednoznačné řešení. V praxi tedy algoritmus neřeší pouze součet nebo součin konkrétní dvojice čísel (např. $6+9$ nebo $6\cdot 9$), ale součet nebo součin obecné dvojice čísel vyjádřených proměnnými *a*, *b*, tj. $a+b$, $a\cdot b$.

1.2.4 Rezultativnost

Algoritmus vede vždy ke správnému výsledku v konečném počtu kroků. Počet kroků může být různý. V některých případech postačuje pár kroků, jindy se jejich počet může blížit nekonečnu.

1.2.5 Efektivnost

Každý krok algoritmu by měl být efektivní. Měl by používat elementární operace, které bude možno provádět v konečném čase.

1.2.6 Numerická stabilita

Algoritmus by měl být navržen jako numericky stabilní s malou, nejlépe žádnou citlivostí na vstupní hodnoty prvků. Případný numerická nestabilita může výsledné údaje zkreslit a závěry z nich vyvozené mohou být chybné. [1]

1.3 Druhy algoritmů

Algoritmy můžeme klasifikovat různými způsoby. Mezi důležité druhy algoritmů patří:

- *Rekurzivní algoritmy*, které využívají (volají) samy sebe.
- *Hladové algoritmy* se k řešení propracovávají po jednotlivých rozhodnutích, která, jakmile jednou učiněna, už nejsou dále revidována.
- *Algoritmy typu rozděl a panuj* dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), po čemž se dílčí řešení vhodným způsobem sloučí.
- *Algoritmy dynamického programování* pracují tak, že postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů. Mnoho úloh se řeší převedením na grafovou úlohu a aplikací příslušného grafového algoritmu.
- *Pravděpodobnostní algoritmy* (někdy též *probabilistické*) provádějí některá rozhodnutí náhodně či pseudonáhodně.
- V případě, že máme k dispozici více počítačů, můžeme úlohu mezi ně rozdělit, což nám umožní ji vyřešit rychleji. Tomuto cíli se věnují *paralelní algoritmy*.
- *Evoluční algoritmy* pracují na základě napodobování biologických evolučních procesů, postupným „pěstováním“ nejlepších řešení pomocí mutací a křížení. V evolučním programování se tento postup aplikuje přímo na algoritmy (resp. programy), které jsou zde chápány jako možná řešení daného problému.
- *Heuristický algoritmus* si za cíl neklade nalézt přesné řešení, ale pouze nějaké vhodné přiblížení. Používá se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

Přitom jeden algoritmus může patřit zároveň do více skupin. Například může být zároveň rekurzivní a typu rozděl a panuj. [3]

2 ANALÝZA VÝKONNOSTI ALGORITMŮ

V této kapitole se pokusíme nalézt odpověď na otázku, který ze dvou algoritmů, jež používají pro řešení stejné otázky rozdílné metody, je účinnější.

Zamyslíme-li se nad problematikou hodnocení efektivity algoritmů, tak nás ihned napadnou minimálně dva důležité faktory.

- Doba realizace
- Zaneprázdnění paměti [4]

2.1 Složitost algoritmu

Složitost je důležitá vlastnost algoritmu, která slouží ke zhodnocení jeho efektivity. Rozeznáváme časovou složitost a paměťovou složitost. Představuje kvantitativní charakteristiku, pomocí které jsou algoritmy srovnávány. Složitost algoritmu definujeme obecně jako vztah algoritmu A k hardwarovým prostředkům H v závislosti na vstupních datech D. Hardwarové prostředky nejsou neomezené, i když jejich výkon neustále roste. Algoritmus by měl být navržen tak, aby poskytoval výsledky v přijatelné době a zbytečně neplýtval hardwarovými prostředky počítače.

Abychom se abstrahovali od závislosti na hardware, vytváříme model *abstraktního* počítače, jehož parametry jsou odvozeny od skutečného počítače. Příkladem abstraktního matematického počítače může být tzv. Turingův stroj.

2.1.1 Časová složitost

Zachycuje dobu běhu algoritmu. Definujeme ji jako funkci, která každé množině vstupních dat přiřadí počet operací, jež musí algoritmus provést. Funkci nazveme časovou funkcí a označíme ji *t*. Počet provedených operací lze ztotožnit s časem *T* nutným pro jejich vykonání. Zpracování vstupních dat *D* algoritmem *A* v čase *T* lze zapsat jako

$$T = t\left(A\left(D\right)\right) \tag{1}$$

Cílem je pro každý algoritmus nalézt takovou funkci *t*, která v závislosti na počtu vstupních dat *N* určí počet operací nutných k vyřešení problému (např.: $t\left(N\right) = 3N^2 + N$)).

Stanovíme-li dobu nutnou pro provedení jedné operace, obdržíme výsledek v časových jednotkách. Základní vlastností abstraktního modelu počítače je předpoklad, že doba

přístupu do paměti je konstantní a okamžitá (tj. nedochází k žádné latenci). Doba běhu je fakticky závislá na použitém hardware, v praxi se hodnoty latence pohybují zpravidla v řádech nanosekund. V tabulce (Tab. 1) jsou uvedeny příklady časových složitostí pro různé hodnoty vstupních dat *N*.

Tab. 1. Ukázky časové složitosti pro různé hodnoty vstupních dat N

Složitost	Vyjádření	N = 10	N = 20	N = 50	N = 100	N = 500
Logaritmická	t(log(N))	1	1,3	1,7	2	2,7
Lineární	t(N)	10	20	50	100	500
Kvadratická	$t(N^2)$	100	400	2500	10000	25000
Kubická	$t(N^3)$	1000	8000	125000	$1,0 . 10^6$	$1,3 . 10^8$
Bikvadratická	$t(N^4)$	10000	160000	$6,3 . 10^6$	$1,0 . 10^8$	$6,3 . 10^{10}$
Exponenciální	$t(2^N)$	1024	$1,0 . 10^6$	$1,1 . 10^{15}$	$1,3 . 10^{30}$	$1,0 . 10^{150}$
Faktoriální	t(N!)	$3,6 . 10^6$	$2,4 . 10^{18}$	$3,0 . 10^{64}$	$9,3 . 10^{157}$	$1,2 . 10^{1134}$

Za efektivní považujeme zpravidla algoritmy se složitostí polynomiální $t\left(N^k\right)$ popř. jejich „kombinace“ $t\left(N^k \cdot \log N\right)$. Tyto algoritmy nejsou příliš citlivé ke změně velikosti vstupních dat, změna velikosti vstupních dat nezpůsobí příliš velkou změnu výpočetního času. Algoritmy s exponenciální nebo faktoriální složitostí jsou naopak velmi citlivé ke změně velikosti vstupních dat, malá změna velikosti vstupních dat může způsobit prodloužení výpočtu i o tisíce let. Algoritmy se složitostí vyšší než polynomickou vedou k tzv. NP problémům. V případě exponenciálních a faktoriálních složitostí není možné dosáhnout významného snížení doby výpočtu zvýšením výkonu počítače, je nutné stávající algoritmus nahradit jiným algoritmem s menší složitostí.

2.1.2 Paměťová složitost

Paměťová složitost, jinak také nazývána prostorová složitost, představuje závislost paměťových nároků algoritmu na vstupních datech. Označujeme ji *S* a definujeme jako maximální množství paměti *M* potřebné k vyřešení algoritmu *A* při počtu *N* vstupních hodnot. Při výpočtu totiž velikost aktuálně používané paměti kolísá, mění se podle momentální potřeby algoritmu. Toto kritérium je používáno méně často než kritérium časové složitosti, objektivně se těžko určuje. Počet obsazených paměťových buněk se

zjišťuje poměrně obtížně. Pro simulaci opět využíváme abstraktní počítač s neomezeným množstvím paměťových buněk. [1]

Mezi časovou a paměťovou složitostí existuje závislost. Na základě znalosti časové složitosti však nelze přímo určit paměťovou složitost a obráceně. Obvykle je však možné zlepšit časovou složitost na úkor prostorové složitosti a naopak. [5]

2.2 Odhady složitosti algoritmů

Přesné stanovení míry složitosti není v řadě případů možné provést nebo je velmi obtížné, je proto možné nahradit ho odhadem (tzv. *asymptotickou složitostí*). Odhad je možné provádět z kroků, u kterých předpokládáme největší složitost. Typickým příkladem jsou různé druhy cyklů či rekurze. Váha ostatních kroků bývá menší, zpravidla ji zanedbáváme. Složitost v tomto případě určujeme zařazením do některé z tříd uvedených v tabulce 1. Je-li složitost vyjádřena např. vzorcem $t(N) = 3N^2 + 4N + 2$, patří algoritmus do třídy algoritmů s kubickou složitostí N^3. Existuje několik typů odhadů složitosti, uveďme tři nejčastěji používané.

2.2.1 Horní odhad složitosti algoritmu

Horní odhad složitosti označujeme *O*. Tohoto odhadu složitosti dosahuje algoritmus pouze ve vzácných případech (tj. skutečná složitost leží v intervalu ohraničeném horním a dolním odhadem složitosti).

Definice: Pro libovolné funkce f, g platí: složitost algoritmu f(N) je asymptoticky menší nebo rovna g(N) tzn. f(N) = O(g(N)), právě když existuje konstanta c, (c > 0), taková, že pro každou libovolnou velikost dat N od N_0, tzn. $N > N_0$, platí

$$0 \leq f(N) \leq c.g(N) \tag{2}$$

Vzorec lze interpretovat takto: funkce *f* tedy neroste rychleji než nějaký c-násobek funkce *g*. Např. zápis $O(N^3)$ udává, že algoritmus má maximálně kubickou složitost. Ve většině případů ji bude mít algoritmus menší, nikdy však větší.

Tento odhad se používá nejčastěji pro hodnocení složitosti algoritmu. Ukazuje největší počet kroků, které je nutno provést k vyřešení zadaného algoritmu. Čím „kvalitnější“ je algoritmus, tím nižší horní odhad má.

2.2.2 Dolní odhad složitosti algoritmu

Dolní odhad složitosti označujeme Ω. Dolní odhad složitosti představuje minimální složitost, jaké algoritmus může dosáhnout. Dosahuje ho pouze zřídka, jedná se o ideální případ. Většinou je skutečná složitost algoritmu větší, pohybuje se v intervalu daném dolním a horním odhadem složitosti.

Definice: Pro libovolné funkce f, g platí: Složitost algoritmu f(N) je asymptoticky větší nebo rovna g(N), tzn. f(N) = Ω(g(N)), právě když existuje konstanta c, (c > 0), taková, že pro každou libovolnou velikost dat N od N_0, tzn. $N > N_0$, platí

$$0 \leq \frac{1}{c} \cdot g(N) \leq f(N) \tag{3}$$

Funkce *g* tedy neroste rychleji než nějaký násobek funkce *f*, tj. *f* roste rychleji než *g*. Uveďme, že definice dolního odhadu se v literatuře liší, tato hodnota se obtížně určuje, jako kvantitativní charakteristika algoritmu se Ω příliš nepoužívá.

Složitost algoritmů nelze neustále snižovat, nemůže klesnout pod teoretickou hodnotu představovanou dolním odhadem složitosti. Algoritmus je považován za optimální v případě, že úlohu řeší s nejmenší možnou složitostí. Taková vlastnost algoritmu nemusí zůstat „na věky“, časem může být nalezen jiný algoritmus, který problematiku řeší efektivněji.

2.2.3 Očekávaná složitost algoritmu

Složitost v průměrném čase bývá označována jako očekávaná složitost a značí se *Θ*. Představuje charakteristiku, která nejlépe charakterizuje skutečnou složitost algoritmu. Určuje se jako střední hodnota (tj. medián) ze složitosti $t(N)$. Může být o několik řádů horší než dolní odhad složitostí a o několik řádů lepší než dolní odhad složitostí. Její praktický výpočet je však poměrně složitý, proto se příliš nepoužívá. Představuje teoretickou hodnotu, snaží se o nalezení nejpravděpodobnější hodnoty „skutečné“ složitosti algoritmu.

Tyto analýzy představují pouze matematický model vztažený k idealizovanému PC. Skutečný výpočetní čas se může měnit v závislosti na HW a SW parametrech stroje. Přesto odhady složitosti představují základní nástroj pro analýzu algoritmu, který moderní informatika používá. K zaručení nízké složitosti algoritmu přispívá také vhodná volba

datových struktur, operace s daty nad těmito strukturami ovlivňují složitost celého algoritmu. [1]

2.3 Praktické porovnávání algoritmů

Porovnání algoritmů lze provádět teoreticky i prakticky. Teoreticky můžeme tedy nalézt odhad na počet kroků algoritmu (časovou složitost) a na spotřebu paměti (paměťovou složitost algoritmu). Často už v teoretických odhadech dostaneme takové rozdíly, že nemá smysl algoritmy dále porovnávat. Pokud je algoritmus složitý, tak může být nalezení správných odhadů obtížné. Nejlepší odhady, kterých jsme schopni mohou být mnohokrát horší než reálné hodnoty. Také záleží na tom, na kterých datech/vstupech budeme algoritmy používat. Z lepší znalosti vstupních dat můžeme ukázat mnohem lepší odhady než pro obecná data. Proto je lepší brát odhady časových složitostí jen jako první a hrubé porovnání. Pro lepší porovnání algoritmů je nezbytné srovnávané algoritmy naprogramovat. Potom můžeme změřit časy výpočtu a velikosti zabrané paměti na používaných datech a ty porovnat.

Dokonce se může stát, že praktickým porovnáním algoritmů dostaneme opačný závěr než teoretickým porovnáním. Tedy že se algoritmus s teoreticky vysokou časovou složitostí může prakticky chovat lépe než algoritmus s teoreticky nízkou časovou složitostí.

Pro praktické porovnání algoritmů je potřeba měřit rychlost na stejném počítači a také na stejných vstupních datech. Také dost záleží na tom, ve kterém programovacím jazyce a jak dobře je který algoritmus implementován. I elegantní algoritmus lze implementovat úplně neelegantně. Čas běhu programu je také značně závislý na hardwaru, na kterém program běží (například drobné zvětšení vstupních dat může výrazně zpomalit běh celého programu, protože se nám najednou data nevejdou do paměti a budou se muset ukládat na disk). Podobná je i závislost na operačním systému.

Pokud už máme dobře proměřené chování algoritmů (jejich časové složitosti) na různě velkých datech, tak je můžeme porovnat. Může se stát, že jeden algoritmus bude lepší pro menší data (například pro $n \leq 100$) a druhý pro větší data (pro $n > 100$). Nastává otázka, který algoritmus vybrat.

Odpověď je nasnadě. Oba. Nejlepšího výsledku dosáhneme kombinací obou algoritmů. Pro malá data použijeme první a pro velká data druhý algoritmus. [6]

3 TŘÍDĚNÍ

Potřeba třídění dat je přímo spojena s lidskou touhou hromadit (chceme vlastnit mnoho) a mít pořádek (když už něco vlastníme, nabízí se otázky, co a kde se vlastně nachází). Třídění velkou mírou přispívá k řešení problému uvedeného na druhém místě. [4]

3.1 Definice problému

Na vstupu je posloupnost $S = (S_1, S_2, ..., S_n)$; cílem je najít takovou posloupnost $S' = (S_1', S_2', ..., S_n')$, pro kterou platí dvě základní kritéria:

- Tato posloupnost je seřazená: $S_1' \leq S_2' \leq ... \leq S_n'$.
- Posloupnost S' je permutací původní posloupnosti S (obsahuje tedy stejná data, jen v jiném pořadí).

V definici relace uspořádání $\leq$ se přitom bere ohled pouze na klíče příslušných hodnot. [7]

3.2 Vysvětlení pojmů

Význam pojmů „třídění“, „řazení“ a „setřídění“ se na základě běžných zkušeností zdá být velmi podobný, ne-li identický. Definujeme neformálně sémantiku těchto pojmů (abstraktních operací nad lineární datovou strukturou) pro potřeby disciplíny programování tak, jak je definována v návrhu revidované názvoslovné normy ČSN.

- *Třídění* (v anglickém jazyce *sorting*, *sort*) je rozdělování údajů na skupiny údajů se stejnými vlastnostmi.
- *Uspořádání pode klíčů* (v anglickém jazyce *collating*) je seřazení údajů podle prvků (klíčů) lineárně uspořádané množiny.
- *Řazení* (v anglickém jazyce *sequencing*) je uspořádání údajů podle relace lineárního uspořádání.
- *Slučování* (v anglickém jazyce *coalescing*) je vytváření souboru sjednocením několika souborů.
- *Setřídění* (nebo také zakládání; v anglickém jazyce *merging*) je vytváření souboru sjednocením několika souborů, jejichž údaje jsou seřazeny podle téže relace uspořádání se zachováním této relace.

Ve smyslu těchto definic budeme algoritmy tradičně nazývané „třídící algoritmy“ nazývat „řadící algoritmy“ a řazení budeme chápat jako zvláštní případ obecnějšího pojmu třídění. V anglickém označení jednotlivých algoritmů však nadále budeme používat jazykový kmen „sort“ ve smyslu řazení tak, jak se používá v anglické terminologii. [8]

4 ŘADÍCÍ ALGORITMY

Řadicí algoritmus je algoritmus zajišťující seřazení daného souboru dat podle specifikovaného pořadí. Nejčastěji se řadí vzestupně nebo sestupně podle numerické velikosti čísel, případně abecedně. Řazení je velmi častá úloha, která je také částí mnoha dalších algoritmů. Vývoji co možná nejefektivnějších algoritmů řazení se proto věnuje velké úsilí.

Z hlediska řazení se vstupní data chápou jako soubor dvojic klíč – hodnota, přičemž po seřazení je posloupnost klíčů monotónní, zatímco na připojené hodnoty se při řazení nebere zřetel a pouze se přesouvají vždy s odpovídajícím klíčem. [7]

4.1 Klasifikace řadících algoritmů

Klasifikaci řadících algoritmů můžeme provádět podle různých kritérií.

4.1.1 Podle typu paměti v níž je řazená struktura uložena

- Metody *vnitřního* (interního) *řazení* se nazývají také metody řazení polí. Předpokládají uložení seřazované struktury v operační paměti a přímý (nesekvenční) přístup k položkám struktury.
- Metody *vnějšího* (externího) *řazení* se nazývají také metody řazení souborů. Předpokládají sekvenční přístup k položkám seřazované struktury. Zvláštní skupinu mohou tvořit struktury s indexsekvenčním přístupem (implementované na magnetických discích, bubnech), u nichž metody řazení často kombinují principy vnitřního i vnějšího řazení.

4.1.2 Podle typu procesoru

- Na sériovém (sekvenčním) procesoru implementujeme *sériové* (sekvenční) *algoritmy řazení*. Myslíme tím, že následující operace algoritmu se může zahájit až po dokončení předcházející.
- Na paralelním procesoru lze implementovat *paralelní algoritmy řazení*. Paralelní procesor umožňuje současný průběh více operací.

4.1.3 Podle stability

- *Stabilní algoritmus řazení* zachovává vzájemné pořadí údajů se shodnými klíči. Stabilita je nutná tam, kde se vyžaduje, aby se při řazení údajů podle klíče s vyšší prioritou neporušilo pořadí údajů se shodnými klíči vyšší priority, získané předcházejícím řazením množiny podle klíčů s nižší prioritou. [8]
- U *nestabilního algoritmu řazení* není zachování vzájemného pořadí prvků se stejným klíčem zaručeno.

4.1.4 Podle velikosti dodatečné pracovní paměti

Kromě samotných řazených dat také algoritmus zpravidla potřebuje nějakou dodatečnou pracovní paměť. Podle velikosti této dodatečné paměti se algoritmy dělí na:

- *Algoritmy, které potřebují pouze konstantní velikost dodatečné paměti* (velikost paměti je nezávislá na množství řazených dat, označuje se jako $O(1)$). Nazývají se jako *algoritmy řazení na původním místě* (*in situ*).
- *Algoritmy, které potřebují dodatečnou paměť*, například místo o velikosti původních dat (tedy $O(N)$ v asymptotickém vyjádření), ve kterém generují seřazený výsledek. [7]

4.1.5 Podle chování na částečně seřazených souborech dat

- *Přirozený řadící algoritmus* je takový, jehož složitost roste respektive klesá v závislosti na míře seřazenosti vstupní posloupnosti – tj. jestliže doba potřebná k seřazení již seřazené podmnožiny údajů je menší, než doba potřebná na seřazení zbývající neseřazené množiny.
- *Nepřirozený řadící algoritmus* je takový, jehož složitost není závislá na míře seřazenosti vstupní posloupnosti.

4.1.6 Podle způsobu využití klíčů

- *Asociativní řazení* (v anglickém jazyce se tyto metody nazývají *comparison sorts*) je takové, když jsou klíče používány pro porovnání vzájemného pořadí prvků

(hodnot klíčů *x* a *y* se využívá pro stanovení relativní polohy ve výstupu, tj. na základě *Klíč (x) <= Klíč (y)*).

- *Adresní řazení* (v anglickém jazyce se tyto metody nazývají *non-comparison sorts*) je takové, když je klíč využit pro výpočet polohy ve výstupu (stanovení absolutní polohy prvku *x* na základě hodnoty *Klíč(x)*). [9]

4.1.7 Podle základního principu řazení

- *Metody pracující na principu výběru* (v anglickém jazyce *selection*) přesouvají postupně největší (nejmenší) prvek ze seřazované množiny do výstupní lineární struktury.
- *Metody pracující na principu vkládání* (v anglickém jazyce *insertion*) zařazují (zatřiďují) do seřazené výstupní lineární struktury postupně všechny prvky seřazované množiny (v libovolném pořadí).
- *Metody pracující na principu rozdělování* (v anglickém jazyce *partition*) rozdělují postupně všechny (pod)množiny na dvě další podmnožiny tak, že všechny prvky jedné podmnožiny jsou menší než všechny prvky druhé podmnožiny.
- *Metody pracující na principu setřídění* (sdružování; v anglickém jazyce *merging*) sjednocují seřazené podmnožiny do větších seřazených podmnožin.
- *Metody pracující na jiných principech* vytváří méně sourodou skupinu různých principů nebo kombinací základních principů.

Dá se říci, že principy výběru a vkládání jsou zvláštními případy principů rozdělení a setřídění, kdy jedna z množin sestává z jediného prvku.

5 ŘAZENÍ NA PRINCIPU VÝBĚRU

5.1 Selection Sort

Tato metoda řazení nevybírá prvky z neseřazené části posloupnosti sekvenčně, ale na základě jejich hodnoty. Prvek je vložen do seřazené části na předem známou pozici, která se již v průběhu třídění nemění. [10]

Pro jednoduchost si představme posloupnost rozdělenou do dvou částí. V jedné části jsou již seřazené prvky, zatímco druhá část je neseřazená. V neseřazené části najdeme nejmenší prvek a přesuneme ho na konec seřazené části (vyměníme s posledním prvkem seřazené části).

5.1.1 Popis algoritmu

- V posloupnosti najdeme nejmenší prvek a vyměníme ho s prvkem na první pozici. Tím dojde k rozdělení posloupnosti na dvě části. Seřazená část obsahuje pouze jeden prvek, neseřazená obsahuje $n-1$ prvků.
- V neseřazené části najdeme nejmenší prvek a vyměníme ho s prvním prvkem v neseřazené části, čímž dojde k zařazení tohoto prvku do seřazené části.
- Obsahuje-li neseřazená část více než jeden prvek, pokračujeme bodem 2, jinak je řazení ukončeno. [9]

5.1.2 Analýza algoritmu

- Minimální, průměrná, maximální asymptotická časová složitost: $O\left(n^2\right)$
- Paměťová složitost: $O(1)$
- Metoda je *přirozená*, *pracuje na místě* (in-situ) a je *nestabilní*. Při výměně minima s i-tým prvkem se *i-tý* prvek může dostat za prvek se shodnou hodnotou klíče a tím se může porušit jejich relativní pořadí ve výsledném seřazeném poli. Metoda však může být implementována i jako stabilní ale za požadavku komplikovanější datové struktury, přičemž metoda ztrácí svou hlavní vlastnost a tou je jednoduchost. [11]
- Selection Sort je jeden z nejjednodušších řadících algoritmů a pracuje velmi dobře pro malé soubory dat. Důležitou vlastností je, že každý prvek je přemístěn

maximálně jednou. Je to metoda, která se hodí pro řazení souborů dat s velmi rozsáhlými prvky (záznamy) a krátkými klíči.

- Nejhorší výsledky podává, pokud je pole dat již setříděno a v sestupném pořadí. [12]
- Mezi jednoduchými algoritmy s časovou složitostí $O(n^2)$, podává Selection Sort lepší výsledky než Bubble Sort nebo Gnome Sort ale je obvykle překonáván Insert Sortem. Na rozsáhlých polích dat je jednoznačně překonáván složitějšími metodami typu rozděl a panuj jako je Quick Sort a Merge Sort. Avšak Selection Sort stejně jako Insert Sort jsou většinou rychlejší na malých polích (méně než 10-20 prvků). Oblíbenou úpravou složitějších algoritmů je tedy použití Selection Sortu nebo Insert Sortu pro dostatečně malá pole prvků. [11]

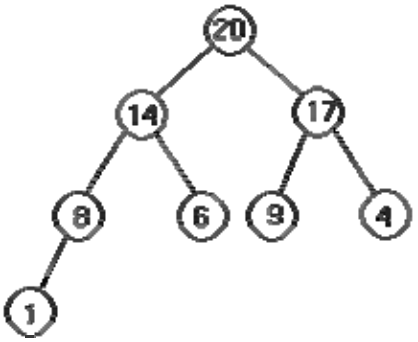
5.2 Heap Sort

Tato metoda je nejvyspělejší metodou postavenou na principu výběru. Zásadním rysem této metody je způsob vyhledání maxima. Za tím účelem definujeme strukturu na základě binárního stromu pro jejíž každý uzel bude platit, že všechny uzly obou podstromů jsou menší než daný uzel. Takovou stromovou strukturu budeme nazývat hromada (angl. heap). [8]

5.2.1 Halda

Halda reprezentující posloupnost S mohutnosti n, je úplný binární strom výšky $h \geq 1$ s n vrcholy, a následujícími vlastnostmi:

- všechny listy se nachází ve vzdálenosti h nebo $h - 1$ od kořene;
- všechny listy na úrovni h jsou vlevo od listů na úrovni $h - 1$;
- každému vrcholu tohoto stromu je přiřazen jeden prvek posloupnosti S tak, že všem jeho potomkům jsou přiřazeny menší prvky.



Obrázek 1: Příklad haldy

Halda se dá výhodně reprezentovat v poli tak, že do pole postupně zapíšeme všechny prvky přiřazené jednotlivým poschodím, zleva doprava, od kořene směrem k listům. Při této reprezentaci platí, že prvek ležící na *i-té* pozici má levého potomka na pozici *2i*, pravého potomka na pozici *2i + 1*. Z podmínek haldy potom plyne, že maximální prvek je vždy v kořenu haldy to znamená na první pozici pole. [21]

Jak se ustaví znovu hromada, nahradí-li se vrchol hromady jiným prvkem? Algoritmu znovu-ustavení hromady se říká „prosetí“ (v anglickém jazyce sift) a název charakterizuje proces, v němž se prvek z vrcholu „propadá“ až na místo, které mu náleží podle pravidla hromady, zatímco maximum (resp. minimum) „vyplave“ na vrchol hromady. [8]

5.2.2 Popis algoritmu

Pole se rozdělí na levou část (haldu) a pravou část (seřazená posloupnost).

- Vlož postupně všechny prvky do haldy (celé pole je halda).
- Odeber největší prvek a přidej ho na začátek seřazené posloupnosti (tzn. přesuň prvek na poslední pozici v haldě a o jedna posuň rozdělení na haldu a seřazenou posloupnost).
- Pokračuj, dokud je halda neprázdná. [22]

5.2.3 Analýza algoritmu

- Minimální, průměrná, maximální asymptotická časová složitost: $O(n \log n)$
- Paměťová složitost: $O(1)$ [17]

- Všechny základní operace s haldou – vložení, zrušení, výměna prvků – vyžadují méně než $2\log n$ porovnání. Všechny tyto operace vyžadují průchod haldou od jejího kořene k listům, což představuje ne více než $\log n$ uzlů pro haldu s *n* prvky. [21]
- Metoda *není přirozená*, *pracuje na místě* (in-situ) a je *nestabilní*. Výsledky jejího časového hodnocení jsou zejména pro větší počet prvků vynikající. [8]
- Co se porovnání s ostatními řadícími algoritmy týče, tak Heap Sort nejčastěji soupeří s metodou Quick Sort. Quick Sort je typicky o něco rychlejší, ale časová složitost Quick Sortu v nejhorším případě je $O\left(n^2\right)$ a ta je neakceptovatelná pro řazení rozsáhlých souborů dat . Heap Sort má horní hranici časové složitosti $O\left(n\log n\right)$ a konstantní paměťové nároky a z tohoto důvodu je často využíván v zapouzdřených systémech s omezeními v reálném čase nebo v systémech starajících se o bezpečnost.
- Heap Sort soutěží také s metodou Merge Sort, která má stejnou časovou složitost v nejhorším případě ale potřebuje $O\left(n\right)$ dodatečné paměti. Heap Sort běží typicky rychleji na počítačích s malou nebo pomalou vyrovnávací pamětí. Na druhou stranu je Merge Sort stabilní, dá se lépe paralelizovat (v té nejjednodušší paralelní implementaci dosahuje Merge Sort téměř lineárního zrychlení, zatímco zatím není jasný způsob jak paralelizovat Heap Sort). Merge Sort lze také snadno upravit pro práci se spojovanými seznamy a velmi rozsáhlými seznamy uloženými na médiích s pomalými přístupovými dobami. Heap Sort je oproti tomu velmi pomalý na médiích s dlouhými přístupovými dobami.

5.2.4 Varianty algoritmu Heap Sort

- Intro Sort
- Ternary Heap Sort
- Smooth Sort

5.3 Intro Sort

Je zajímavou alternativou k metodě Heap Sort. Kombinuje výhody algoritmů Quick Sort i Heap Sort tj. maximální časovou složitost Heap Sortu a průměrnou časovou složitost Quick Sortu. [23]

5.4 Ternary Heap Sort

Ternary Heap Sort používá ternární haldu místo binární haldy. Znamená to, že každý vrchol v haldě má tři potomky. Oproti Heap Sortu je tento algoritmus složitější naprogramovat ale provádí několikrát méně operací porovnání a přesunu prvků. [23]

5.5 Smooth Sort

Tato metoda má stejně jako Heap Sort horní odhad časové složitosti $O(n \log n)$. Výhodou Smooth Sortu je však to, že horní odhad se snižuje blíže k $O(n)$ pokud je vstupní posloupnost už v určitém stupni seřazena. Kvůli své složitosti je tato metoda používána velmi zřídka. [23]

5.6 Bubble Sort

Tento algoritmus je odvozen od chování vzduchových bublinek v limonádě, kterou postupně stoupají vzhůru k hladině. [10]

Algoritmus řadí přímou výměnou sousedních prvků. Procházíme polem, porovnáváme dva sousední prvky a pokud nejsou v požadovaném pořadí, prohodíme je.

5.6.1 Popis algoritmu

- Posloupnost rodělíme na dvě části, seřazenou a neseřazenou. Seřazená část je prázdná.
- Postupně porovnáme všechny sousední prvky v neseřazené části a pokud nejsou v požadovaném pořadí, prohodíme je.
- Druhý krok opakujeme tak dlouho, dokud neseřazená část obsahuje více než jeden prvek. Jinak algoritmus končí.

- Modifikace (*Přirozený Bubble Sort*): Pokud nastala ve druhém kroku alespoň jedna výměna, došlo ke zmenšení nesetříděné části o jeden prvek, tak pokračujeme druhým krokem. Jinak je algoritmus ukončen. [9]

5.6.2 Analýza algoritmu

- Průměrná, maximální asymptotická časová složitost: $O(n^2)$, minimální asymptotická časová složitost: $\Omega(n)$
- Paměťová složitost: $O(1)$
- Metoda se chová *stabilně* a *přirozeně* a *pracuje na místě*. Je ze všech metod nejrychlejší, řadí-li se již seřazené pole. Její princip je vhodný pro testování, zda je pole seřazené. V jiných případech je nejméně účinnou metodou. [8]
- Bubble Sort má stejnou asymptotickou časovou složitost v nejhorším případě jako Insertion Sort, ale tyto dva algoritmy se výrazně liší v počtu nutných prohození prvků. Insertion Sort jich potřebuje výrazně méně.
- Bubble Sort se také chová špatně při použití moderních CPU. Potřebuje nejméně dvakrát tolik zápisů a má dvakrát více chyb ve vyrovnávací paměti. [13]

5.6.3 Varianty algoritmu Bubble Sort

- Ripple Sort
- Shaker Sort
- Shuttle Sort

5.7 Ripple Sort

Tato varianta této metody si pamatuje pozici první dvojice u které došlo k výměně. V příštím cyklu začíná porovnávat až od předcházející dvojice.

5.8 Shaker Sort

Varianta zvaná Shaker Sort prochází pole střídavě zleva-doprava a zprava-doleva. Seřazené části pole jsou v průběhu řazení na obou koncích pole a při ukončení řazení se

spojí. Tento postup připomíná práci barmana se shakerem, kterým pohybuje nahoru a dolů, proto algoritmus dostal název shaker sort.

5.9 Shuttle Sort

Varianta zvaná Shuttle-Sort pracuje tak, že dojde-li u dvojice k výměně (např. při průchodu zleva-doprava), vrací se metoda s prvkem, který se posunuje doleva tak dlouho, dokud dochází k výměně. Pak se vrací do pozice u níž ukončila posun doprava a pokračuje směrem vpravo. Metoda končí, porovná-li úspěšně poslední dvojici prvků. [8]

5.10 Gnome Sort

Gnome Sort je řadící algoritmus podobný metodě Insertion sort kromě toho, že k zařazení prvku na správné místo používá sérii výměn, stejně jako metoda Bubble Sort. Jméno metody je inspirováno chováním holandského zahradního trpaslíka, který má za úkol seřadit řadu květináčů. [19]

5.10.1 Popis algoritmu

- Aktuální prvek porovnáváme s předchozím prvkem.
- Pokud jsou ve správném pořadí, tak se přesuneme na další prvek (nebo zastavíme, pokud jsme již dosáhli konce posloupnosti).
- Pokud nejsou ve správném pořadí, tak prvky prohodíme a přesuneme se na předchozí prvek.
- Pokud není žádný předchozí prvek, tak se přesuneme na další prvek. [20]

5.10.2 Analýza algoritmu

- Maximální asymptotická časová složitost: $O\left(n^2\right)$, minimální asymptotická časová složitost: $\Omega\left(n\right)$ (pokud je posloupnost na začátku téměř setříděna)
- Paměťová složitost: $O\left(1\right)$
- Metoda se chová *stabilně* a *pracuje na místě*. Je to velmi jednoduchý algoritmus, který neobsahuje žádnou vnořenou smyčku. [19]

6 ŘAZENÍ NA PRINCIPU VKLÁDÁNÍ

6.1 Insertion Sort

Tento způsob řazení připomíná řazení karet. Postupně vybíráme jednu kartu za druhou a zařazujeme je na odpovídající místo podle barvy a hodnoty - klíče, ostatní ještě nezařazené posuneme o jednu pozici.

6.1.1 Popis algoritmu

- První prvek pole ponecháme na svém místě.
- Vezmeme druhý prvek a porovnáme jej s prvním. Je-li menší, zařadíme ho na první místo a první prvek posuneme, jinak je ponecháme na místě.
- Vezmeme třetí prvek a porovnáme jej s prvními dvěma prvky. Je-li menší než některý z nich, zařadíme jej na odpovídající pozici a následující prvky podle potřeby posuneme. Jinak je ponecháme na původních místech.
- Obdobně postupujeme i s ostatními prvky v poli. [9]

6.1.2 Analýza algoritmu

- Průměrná, maximální asymptotická časová složitost: $O\left(n^2\right)$, minimální asymptotická časová složitost: $\Omega\left(n\right)$
- Paměťová složitost: $O\left(1\right)$
- Metoda je *stabilní*, chová se *přirozeně* a *pracuje na místě*.
- Je efektivní na malých množinách prvků a efektivní na částečně seřazených množinách (běží v čase $O\left(n+d\right)$, kde *d* je počet transpozic prvků množiny).
- Je efektivnější než většina ostatních $O\left(n^2\right)$algoritmů (Selection sort, Bubble sort), průměrný čas je $N^2/4$ a v nejlepším případě je dokonce lineární.
- Je to online algoritmus, dokáže řadit data tak, jak přicházejí na vstupu. [14]

6.1.3 Varianty algoritmu Insertion Sort

- Binary Insertion Sort
- Binary Tree Sort
- Library Sort
- Shell Sort

6.2 Binary Insertion Sort

Tato varianta metody řazení vkládáním používá k vyhledání pozice pro zařazovaný prvek mechanismus binárního vyhledávání. Binární vyhledávání musí v případě rovnosti klíčů nalézt pozici za nejpravějším ze shodných klíčů. [8]

6.3 Binary Tree Sort

Je to metoda, která používá pro seřazení klíčů datovou strukturu *binární vyhledávací strom* a následně manipuluje se stromem tak, že jsou klíče seřazeny.

Při použití struktury samo-vyvažujícího binárního vyhledávacího stromu je metoda stabilní a má tyto parametry:

- Průměrná, maximální asymptotická časová složitost: $O(n \log n)$
- Paměťová složitost: $O(n)$ [15]

6.4 Library Sort

Metoda Library Sort, která je také nazývána *Gapped Insertion Sort* je řadící algoritmus, který používá metodu Insertion Sort ale s prázdnými mezerami v poli za účelem zrychlení následujících vkládání prvků. Jméno metody vychází z následující analogie:

Předpokládejme, že knihovník má za úkol poskládat knihy, na dlouhou polici, seřazené podle abecedy. Na levém konci police řadí knihy začínající písmenem *A* a postupuje v řazení knih doprava po polici, aniž by mezi knihami nechával jakékoliv mezery až ke konci police, kde skládá knihy začínající písmenem *Z*. Kdyby knihovník pořídil novou knihu, jejíž název bude začínat písmenem *B*, tak musí knihovník nalézt správnou pozici pro knihu mezi knihami, které začínají na písmeno *B*. Následně bude muset pohnout

doprava se všemi knihami začínajícími písmenem *B* až po ty knihy, které začínají písmenem *Z*, aby udělal prostor pro zařazení nové knihy doprostřed sekce knih, které začínají na písmeno *B*. Toto je princip Insertion Sortu.

Kdyby však knihovník nechával nějaký volný prostor za každým písmenem, tak by musel pohnout vždy jen pár knihami, aby zařadil knihu začínající na písmeno *B* doprostřed sekce knih začínajících písmenem *B*, dokud by stále bylo nějaké volné místo za sekcí knih začínajících na písmeno *B*.

6.4.1 Analýza algoritmu

- Průměrná asymptotická časová složitost: $\Theta(n \log n)$, maximální asymptotická časová složitost: $O(n^2)$
- Paměťová složitost: $O(n)$
- Metoda je *stabilní*.
- Je to online algoritmus. [16]

6.5 Shell Sort

Algoritmus řazení se snižujícím se přírůstkem, označovaný podle jména autora názvem Shell Sort, vzbudil při svém uvedení v 60.letech velkou pozornost. Ještě v publikaci o 10 let později se uvádí: cituji „Bohužel, není snadné pochopit činnost této metody. Když se poprvé objevila v tisku, jistý vedoucí programátor a systémový programátor, kteří nemohli pochopit její postup vytvořili program a podrobili metodu řadě pokusů, v nichž dobře obstála. Přesto, že stále nechápali její tajuplný postup, zařadili ji do knihovny programů. ...Pokud se některý z čtenářů domnívá, že by porozuměl Shellově metodě, může to zkusit.“ Je nutné podotknout, že dále v publikaci uvedený algoritmus vyjádřený vývojovým diagramem je opravdu téměř nepochopitelný do okamžiku, než se přepíše podle zásad strukturovaného programování, kdy je srozumitelný bez komentářů i pro průměrného studenta.

V čem spočívá princip Shellovy metody? Metoda využívá některé z jednoduchých řadících metod s kvadratickou asymptotickou časovou složitostí $O(n^2)$ (nejvhodnější je právě metoda Insertion Sort pro své kvality u řazení téměř setříděného pole). V těchto metodách

se prvky porovnávají a přemisťují vždy o 1 pozici. Lze přitom dokázat, že se každý prvek ve středním případě přesune celkem o *n/3* pozic. Mechanismus řazení by se dal urychlit, kdyby se jednoduchá metoda mohla pohybovat v poli většími skoky.

6.5.1 Popis algoritmu

- Prvky rozdělíme na 4 skupiny tak, že prvky každé skupiny jsou od sebe vzdáleny o krok $h_3 = 4$ a každou skupinu zvlášť seřadíme vhodnou jednoduchou metodou.
- V další etapě rozdělíme všechny prvky pole na 2 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 by sama o sobě seřadila pole prvků a kdyby se jako metoda řazení použila metoda, která se nechová přirozeně, byly by "předzpracující" etapy něčím navíc! Skutečnost však ukazuje, že "předřazovací" etapy urychlí přesun prvků na svá místa a způsobí, že poslední etapa proběhne mnohem rychleji.

Shellova metoda nebyla dosud důkladně matematicky analyzována (a protože byla výrazně překonána metodami s časovou složitostí $O(n \log n)$, praktický zájem o tuto metodu poklesl), a proto není matematicky rozhodnuto, posloupnost jakých snižujících se kroků h_s, h_s-1, ..., h_1 je nejvýhodnější. Mezi nejznámějšími návrhy byly tyto posloupnosti :

A. h_1=1, $h_{i+1} = 2h_i + 1$

B. $h_1 = 1$, $h_2 = 3$, $h_{i+1} = 2h_i - 1$ (pro $i > 2$)

C. $h_1 = 1$, $h_{i+1} = 3h_i + 1$ (kde nejvyšším použitým h_i je nejmenší h_i, pro něž platí h_{i+2}>=n)

Pro posloupnost B byla dokázána maximální časová složitost $O(n^{1,5})$, což znamená, že Shellova metoda zaujímá pozici mezi pomalými $O(n^2)$ a rychlými $O(n \log n)$ metodami řazení. [8]

6.5.2 Analýza algoritmu

- Maximální asymptotická časová složitost: $O(n \log^2 n)$

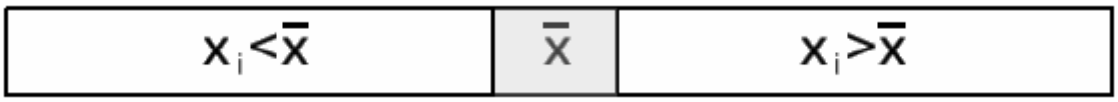
- Paměťová složitost: $O(1)$ [17]
- Metoda *není stabilní* (což je důsledkem kroku většího než 1) a *pracuje na místě*. Tvrzení o přirozenosti není jednoznačné. [8]
- Shell Sort je vhodná metoda pro mnoho aplikací řazení z důvodu akceptovatelné rychlosti dokonce pro středně rozsáhlé soubory a jednoduché implementace algoritmu. [18]

7 ŘAZENÍ NA PRINCIPU ROZDĚLENÍ

7.1 Quick Sort

Tento algoritmus je jedním z nejpoužívanějších řadících algoritmů. Využívá principu rozděl a panuj, který v tomto případě způsobuje výrazné urychlení řadícího procesu. Vysokého výkonu dosáhne prohazováním prvků na velké vzdálenosti.

Ideu algoritmu Quick Sort lze nejlépe ilustrovat na následujícím příkladu. Máme posloupnost prvků seřazených sestupně, tj. pro $\forall i \in \langle 1, n \rangle$ platí $x_i \geq x_{i+1}$. Pokud bychom chtěli takovou posloupnost seřadit vzestupně, jako velmi rychlý by se jevil následující postup prohazování: $x_1 \Leftrightarrow x_n, x_2 \Leftrightarrow x_{n-1}, ..., \ldots$ tj. výměna prvního prvku s posledním, druhého s předposledním, atd. V takovém případě nebude potřeba k setřídění *n-1* výměn, ale pouze *n/2*. [1]



Obrázek 2: Rozdělení posloupnosti vzhledem k pivotu $\bar{X}$

7.1.1 Popis algoritmu

- V prvním kroku je zvolena střední hodnota *x*, tzv. *pivot*.
- Posloupnost je uspořádána rozdělením na dvě části tak, že levá část obsahuje prvky $x_i < x$, pravá část prvky $x_i > x$.
- Hodnota pivotu x může být zvolena náhodně, představovat medián či aritmetický průměr (viz podkapitola 7.1.2).
- Každá z těchto částí je stejným způsobem rekurzivně rozdělena. Počty úseků se s každým dělením zdvojnásobují, délky úseků (tj. počty prvků) snižují na polovinu.
- Dělení provádíme tak dlouho, dokud délky úseků nejsou rovny jedné. V takovém případě je posloupnost seřazena. [10]

7.1.2 Volba pivota

První krok představuje volbu pivota. Pivota je možno volit jako:

- $\overline{X} = X_1$

  Pivot je představován prvním prvkem z posloupnosti. Jedná se o nejméně výhodnou volbu, postup řazení je velmi pomalý. Lze zobecnit, že pro jakoukoliv pevnou polohu pivota v každém dělení je algoritmus Quick Sort pomalý.

- *Medián*

  Nejlepší varianta, algoritmus dosahuje nejvyššího výkonu. Problém představuje nalezení mediánu, složitost tohoto postupu činí $O(n)$. Výsledkem je rychlý postup řazení, do celkové složitosti algoritmu je nutné zahrnout i proces hledání mediánu.

- *Náhodný prvek*

  Nejčastěji používaná varianta., složitost algoritmu je $O(n \log n)$. Algoritmus je ve většině případů téměř tak rychlý, jako při použití mediánu. [1]

7.1.3 Analýza algoritmu

- Průměrná, minimální asymptotická časová složitost: $O(n \log n)$, maximální asymptotická časová složitost: $O(n^2)$

- Paměťová složitost: $O(n)$ [8]

- Metoda *není stabilní* (při vzájemné výměně se může změnit relativní pořadí shodných klíčů). Jak je vidět z experimentů, lze ale odvodit i z principu, metoda *nepracuje přirozeně* (seřazení náhodně uspořádaného pole je téměř dvojnásobně kratší oproti seřazení již seřazeného nebo opačně seřazeného pole). Metoda v důsledku rekurze *nepracuje in situ*. Tyto vlastnosti však nebrání tomu, aby metoda byla hodnocena jako jedna z nejúspěšnějších řadících metod, vhodných pro větší počty prvků. V další části kapitoly bude uvedena modifikace této metody spočívající v nahrazení rekurzivního zápisu algoritmu iterací se zásobníkem. [8]

- Praktické testy ukázaly, že Quick Sort je velice rychlý při řazení rozsáhlých polí, ale zaostává oproti přirozeným algoritmům (Insert Sort, Select Sort) při řazení malých polí. Do jistého počtu prvků má Quick Sort větší režii (řadí pomaleji) než např. Insertion Sort. Tato hranice byla experimentálně stanovena na asi 12 prvků. Vyplatilo by se tedy Quick Sortem řadit rozsáhlé pole, ale jakmile úseky na něž se

pole dělí budou kratší než zvolená mez, tak tento krátký úsek dořadit Insertion Sortem. [21]

- Přestože Quick Sort nemá zaručenou časovou složitost $O(n \log n)$, reálné aplikace a testy ukazují, že na pseudonáhodných datech je vůbec nejrychlejší ze všech obecných řadících algoritmů (tedy i rychlejší než Heap Sort a Merge Sort, které jsou formálně rychlejší). Maximální časová náročnost $O(n^2)$ ho však diskvalifikuje pro použití v kritických aplikacích. [24]

7.1.4 Varianty algoritmu Quick Sort

- Non-recursive Quick Sort

7.2 Non-recursive Quick Sort

Z hlediska praktického využití jedné z nejrychlejších metod je významnou modifikací odstranění rekurze se současnou redukcí paměťové složitosti.

Mechanismus rozdělení rozdělí daný úsek pole na dva podúseky. Pro jeden z nich se může v iteraci znova vyvolat mechanismus rozdělení, zatímco hraniční indexy druhého se uchovají v zásobníku pro pozdější zpracování. Místo abychom systematicky levý podúsek dělili a pravý uchovávali v zásobníku, budeme dělit vždy menší podúsek, zatímco meze většího uchováme v zásobníku. Jakmile má menší z podúseků méně než 2 prvky, vybereme úsek pro další dělení ze zásobníku. V dělení pokračujeme, pokud je zásobník neprázdný. Bude-li se dělit vždy menší podúsek, pak v nejhorším případě (budou-li oba podúseky vždy stejně dlouhé) bude potřebná kapacita zásobníku $k = \log_2 n$.

Základní vlastností tohoto algoritmu jsou stejné jako pro rekurzivní verzi. Pro paměťovou složitost však platí : $O(n \log n)$

Z toho vyplývá, že nerekurzivní verze s volbou děleného podúseku je paměťově výrazně výhodnější než rekurzivní verze. Výsledky (čas) nerekurzivní verze jsou při značné úspoře paměti nevýznamně horší. [8]

8 ŘAZENÍ NA PRINCIPU SETŘÍDĚNÍ

8.1 Merge Sort

Základní idea Merge Sortu spočívá v dělení původní posloupnosti na dvě části (nejlépe o poloviční počtu prvků), jejich setřídění a poté použití metody slučování. Výsledkem je setříděná posloupnost o stejném počtu prvků jako byl v původní posloupnosti. Rekurzivně pak obě posloupnosti setřídíme rozdělením na dvě části a slučováním setříděných posloupností.

Rozdělování posloupnosti na části skončí pokud bude posloupnost setříděná. Nejmenší setříděnou posloupností je posloupnost jednoprvková. Dělení se obvykle ukončuje při takovém počtu prvků, který je možné setřídit některou metodou vnitřního třídění.

Každé rekurentní volání znamená rozdělení posloupnosti na dvě části a návrat zpět znamená slučování rozdělených (již setříděných) částí do jedné pomocí metody slučování. [21]

8.1.1 Popis algoritmu

- Rozděl posloupnost na dvě poloviny.
- Rekurzivně setřiď každou polovinu.
- Pokud je posloupnost velikosti jedna, je již setříděná.
- Sluč obě poloviny dohromady. [22]

8.1.2 Analýza algoritmu

- Průměrná, minimální, maximální asymptotická časová složitost: $O(n \log n)$
- Paměťová složitost: $O(n)$
- Metoda je *stabilní*, *chová se přirozeně* a *nepracuje in situ*, protože potřebuje dvojnásobně dlouhé pole, než je seřazované pole.
- Velkou nevýhodou oproti algoritmům stejné rychlostní třídy (např. Heap Sort) je, že Merge Sort pro svou práci potřebuje navíc pole o velikosti *n*. Existuje sice i modifikace Merge Sortu (tzv. *In-place Merge Sort*), která toto pole nepotřebuje, ale

její implementace je velmi složitá a díky vysokému overheadu je i pomalá. Kromě toho se Merge Sort také ukazuje být pomalejší než Quick Sort nebo Heap Sort. Na druhou stranu je Merge Sort stabilní řadící algoritmus, lépe se paralelizuje a má vyšší výkon na sekvenčních médiích s nižší přístupovou dobou. V mnoha implementacích programovacích jazyků je Merge Sort implicitním řadícím algoritmem (v Perlu 5.8, v Javě nebo v GNU C Library). [25]

9 ŘAZENÍ NA JINÝCH PRINCIPECH

Následující kapitola popisuje především algoritmy využívají adresní řazení. Adresní řadící algoritmy nepoužívají při své činnosti žádnou preferovanou operaci (na rozdíl od výše popisovaných algoritmů používajících asociativní řazení, jejichž preferované operace jsou především porovnání a také přesun prvků).

9.1 Bucket Sort

Metoda Bucket Sort, nebo také nazýván Bin Sort je řadící algoritmus, jehož princip je takový, že rozdělí pole do určitého počtu přihrádek. Každá přihrádka je potom řazena individuálně, buď použitím jiného řadícího algoritmu, a nebo rekurzivně metodou Bucket Sort. Bucket Sort je *zobecněním metody Pigeonhole Sort*. [27]

9.1.1 Popis algoritmu

- Vyrob „přihrádky“ pokrývající celý rozsah prvků.
- Projdi posloupnost a každý prvek dej do příslušné přihrádky.
- Setřiď každou neprázdnou přihrádku.
- Projdi přihrádky a dej prvky zpátky do pole. [22]

9.1.2 Analýza algoritmu

- Průměrná asymptotická časová složitost: $\Theta(n.k)$, maximální asymptotická časová složitost: $O(n^2.k)$
- Paměťová složitost: $O(n.k)$ [17]
- Metoda je *stabilní a nepracuje na místě*.
- Bucket Sort může být vnímán jako zobecnění metody Counting Sort. Ve skutečnosti pokud má každá přihrádka velikost 1, tak potom Bucket Sort „zdegeneruje“ v Counting Sort.
- Bucket Sort se dvěmi přihrádkami je verze Quick Sortu, kde hodnota pivota je vždy medián. [27]

9.2 Counting Sort

Metoda Counting Sort (stejně jako Bucket Sort) těží z toho, že dopředu zná rozsah (tj. minimální a maximální hodnotu) čísel z pole, které má být seřazeno (pole *A*). Znalost toho rozsahu použije k vytvoření pole *C*. Každý index *I* v poli *C* je potom použit ke spočtení, kolik prvků z pole *A* má hodnotu *I*. Tyto počítadla uložená v poli *C* jsou následně použity ke vložení prvků v poli *A* na správnou pozici ve výsledném seřazeném poli. [26]

9.2.1 Analýza algoritmu

- Průměrná, maximální asymptotická časová složitost: $O\left(n+2^k\right)$ (kde *n* je délka pole *A* (vstupní pole) a *k* je délka pole *C* (pole pro počítání)).
- Pokud má být algoritmus efektivní, tak hodnota k nesmí být mnohem větší než hodnota *n*.
- Paměťová složitost: $O\left(n+2^k\right)$ [17]
- Metoda je *stabilní*.
- Metoda se používá pouze pro řazení celých čísel. [26]

9.3 Radix Sort

Radix Sort je řadicí algoritmus, který řadí celá čísla postupným procházením všech číslic. Jelikož celočíselné hodnoty mohou reprezentovat řetězce (jména, data apod.), a dokonce i vhodně formátovaná čísla s plovoucí desetinnou čárkou, Radix Sort není omezen pouze na řazení celých čísel.

Většina digitálních počítačů vnitřně reprezentuje všechna data jako binární čísla, nejpřirozenější je pro něj tedy řazení podle skupin bitů (tj. podle číslic o základu 8, 16, 32, 256 apod.). Radix Sort má dvě verze a to LSD (Least Significant Digit) a MSD (Most Singificant Digit). LSD Radix Sort zpracovává reprezentace celých čísel od nejméně významné číslice a pohybuje se směrem k nejvíce významné číslici. MSD pracuje naopak.

Reprezentace celých čísel, které jsou zpracovávány řadícími algoritmy jsou nazývány klíče. Tyto klíče mohou existovat samostatně nebo být asociovány k jiným datům. LSD Radix Sort obvykle používá následující řadící princip: krátké klíče jsou před delšími klíči a

klíče stejné délky se řadí lexikograficky. Toto se shoduje s normálním řádem celých čísel. MSD radix sort používá lexikografické třídění, které je vhodné pro třídění řetězců, nebo celých čísel pevné délky. [28]

Klíč lze reprezentovat číslem jistého rozsahu. Radix Sort neporovnává dva klíče, nýbrž zpracovává a porovnává části klíčů. Radix Sort považuje klíče za čísla zapsaná v číselné soustavě o základu M (*radix*) a pracuje s jednotlivými číslicemi. [21]

9.3.1 Popis algoritmu LSD Radix Sort

- Vezmi nejméně významnou číslici každého klíče.
- Seskup klíče založené na té číslici (LSD), ale také zachovej originální pořadí klíčů (Tento krok dělá LSD Radix Sort stabilním.)
- Opakuj seskupující proces s každou další více významnou číslicí.

Řazení v kroku 2 je obvykle prováděno Bucket Sortem nebo Counting Sortem, které jsou účinné v případě, že je zde pouze malé množství číslic, což obvykle je.

9.3.2 Popis algoritmu MSD Radix Sort (rekurzivního)

- Vezmi nejméně významnou číslici každého klíče.
- Seřaď seznam prvků založených na této číslici, seskupení prvků se stejnou číslicí do jedné přihrádky.
- Rekurzivně seřaď každou přihrádku, začni s další číslicí vpravo.
- Pospojuj přihrádky dohromady ve správném pořadí. [28]

9.3.3 Analýza algoritmu

- Průměrná, maximální asymptotická časová složitost: $O(n.k)$
- Paměťová složitost: $O(n)$ [17]
- Metoda je *stabilní (MSD není), nechová se přirozeně* a *nepracuje na místě.* [8]
- LSD Radix sort nepracuje na místě což znamená, že klíče nebo ukazatele na klíče musí být dočasně uloženy v dodatečné paměti během řazení.

- U stejně dlouhých klíčů, potřebuje LSD Radix sort jeden řadicí průchod vstupní posloupností pro každou číslici v klíči.
- Ani jedna verze Radix Sortu není účinná pokud jsou data již na začátku seřazena, protože obě verze tento fakt ignorují a data seřadí znovu. Naštěstí může být kompletně seřazená posloupnost jednoduše zjištěna v prvním průchod daty, když se rozhoduje o velikosti přihrádek. [28]

9.4 Bogo Sort

Bogo Sort uvádím jako zástupce tzv. nepraktických algoritmů.

V počítačové vědě je Bogo Sort (nebo také nazýván Random Sort) velmi neefektivní algoritmus. Používá se pouze k výukovým účelům, tedy dává se do kontrastu s mnohem účinnějšími algoritmy (popisovanými výše).

9.4.1 Popis algoritmu:

Kdybychom jej použili na setřídění balíčku karet, tak by sestával z následujících kroků:

- Zkontroluj zda je balíček setříděn. Pokud ano, tak konec.
- Pokud není tak vyhoď všechny karty z balíčku do vzduchu.
- Seber karty náhodně ze země a opakuj proces dokud nebude balíček seřazen.

9.4.2 Analýza algoritmu

- Maximální asymptotická časová složitost: ∞, Průměrná asymptotická časová složitost: $O(n.n!)$
- Paměťová složitost: $O(1)$
- Metoda je *nestabilní*, *nechová se přirozeně* a *pracuje na místě*. [34]

10 PRINCIPY VNĚJŠÍCH SEKVENČNÍCH ŘADÍCÍCH METOD

Základním principem řadících metod na vnějších, sekvenčně organizovaných paměťových médiích, je setřiďování (merging). Seznámíme se stručně se základními myšlenkami několika metod. Pro další úvahy budeme předpokládat, že seřazovanou datovou strukturou je soubor.

10.1 Metoda přímého setřiďování (Straight Merging)

Tato metoda potřebuje kromě souboru, v němž jsou seřazovaná data, dva další pomocné soubory.

V prvním kroku metody se data původního (zdrojového) souboru rozdělí rovnoměrně do dvou pomocných souborů. Ve druhém kroku se přečte z každého pomocného souboru jeden prvek, vytvoří se uspořádaná dvojice, která se uloží do zdrojového souboru. Tato akce se opakuje až se ze všech údajů vytvoří soubor uspořádaných dvojic. Ve třetím kroku se uspořádané dvojice rozdělí rovnoměrně do dvou pomocných souborů. Ve čtvrtém kroku se setříděním ze dvou souborů seřazených dvojic vytvoří soubor seřazených čtveřic atd. Tento cyklus se opakuje pro všechny 2^k-tice až po k, pro něž platí, že $2^{k+1}>=N$. Jako poslední setřídíme poslední 2^k-tici a zbývající seřazenou posloupnost ve druhém souboru. Protože typickou implementací vnějšího sekvenčního souboru je magnetická páska, říká se této metodě také přímá metoda tří pásek.

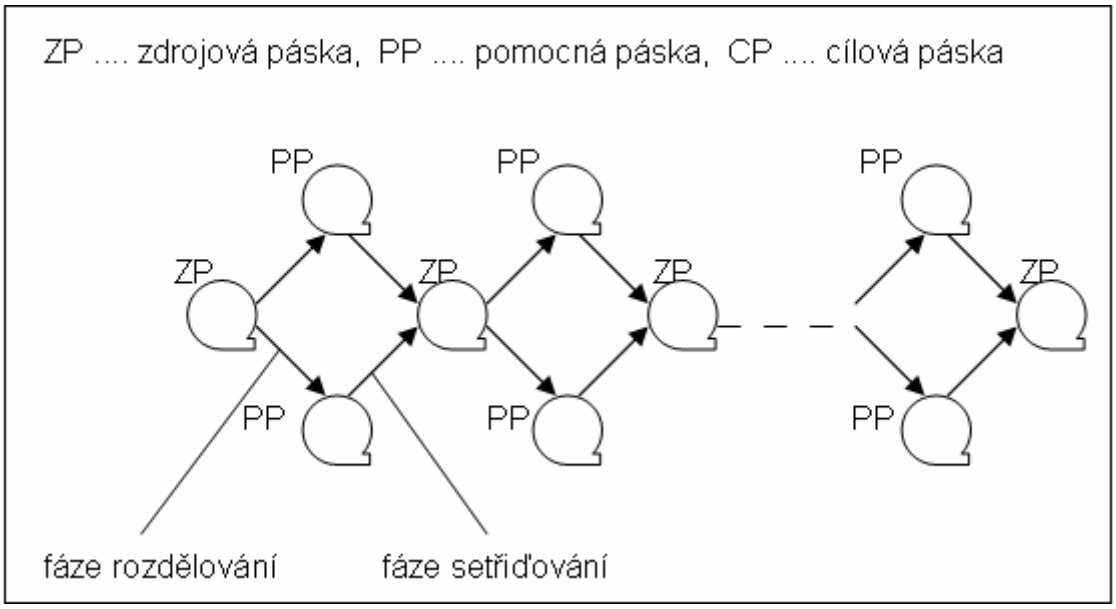
10.2 Metoda přímého vyváženého setřiďování (Straight Balanced Merging)

Jednoduchou variantou předchozí metody přímého setřiďování je přímá metoda čtyř pásek, která za cenu dalšího pomocného souboru (pásky) zkracuje proces vyloučením etapy rozdělení tím, že setříděné 2^n-tice se střídavě ukládají do jednoho ze dvou „cílových“ souborů. Jakmile se „zdrojové“ soubory vyčerpají, provede se záměna funkcí dvojic souborů. „Cílové“ pásky se stanou „zdrojovými“ a „zdrojové“ se stanou „cílovými“.

10.3 Metoda přirozeného setřiďování (Natural Merging)

Na rozdíl od předchozí přímé metody tří pásek, přirozená metoda tří pásek nevytváří setříděné posloupnosti o délce 2^n při vzrůstajícím n, ale využívá počátečního uspořádání

seřazovaného souboru. V první fázi rozdělování zapisuje střídavě do dvou pomocných souborů celé neklesající posloupnosti. Dále se již střídá fáze setřiďování a rozdělování s tím, že při setřiďování musí algoritmus rozeznat konec neklesající posloupnosti. Proces končí setříděním posledních dvou posloupností do zdrojového souboru.



Obrázek 3: Schéma postupu třípáskových metod

10.4 Metoda přirozeného vyváženého setřiďování (Natural Balanced Merging)

Tato metoda podobně jako u přímého setřiďování, zkracuje proces řazení vyloučením fází rozdělení za cenu použití dalšího pomocného souboru (pásky).

Nelze-li u čtyřpáskové metody umístit v závěrečném setřiďování výsledný seřazený soubor na původní zdrojovou pásku, vloží se do procesu řazení kopírovací cyklus, který umístí seřazený soubor na původní pásku.

10.5 Metoda mnohacestného vyváženého setřiďování (Balanced Multiway Merging)

Doba potřebná k seřazení sekvenčního souboru je úměrná počtu potřebných fází, kde každá fáze obsahuje průchod všemi daty v seřazovaném souboru. Jedním ze způsobů, jak snížit počet průchodů spočívá v rozdělení posloupnosti do více než dvou souborů s následným setřiďováním více zdrojových souborů do cílových souborů. Obsahuje-li

seřazovaný soubor r neklesajících posloupností, pak jejich rovnoměrné rozdělení do N souborů a následné setřídění vytvoří r/N posloupností. Ve druhé fázi vznikne r/N^2 a k-té fázi r/N^k posloupností. Celkový počet přepisovacích operací pro N-cestné setřiďování (vyžadující $2N$ souborů) je v nejhorším případě dán vztahem $M = n\lceil \log_N n \rceil$, kde n je počet prvků seřazovaného souboru.

Na rozdíl od dvoucestného vyváženého setřiďování (přirozená metoda 4 pásek) mnohacestné setřiďování končí v jedné fázi, až když jsou vyčerpány neklesající posloupnosti všech souborů. [8]

11 JAZYKY C, C++ A KNIHOVNA WXWIDGETS

11.1 Jazyk C

Byl vymyšlen a poprvé implementován Dennisem Ritchiem na počítači DEC PDP-11 pod operačním systémem UNIX. C je výsledkem vývojového procesu, který odstartoval starší jazyk nazvaný BCPL vyvinutý Martinem Richardsem. BCPL ovlivnil jazyk nazvaný B, který byl vytvořen Kenem Thompsonem a který vedl v 70. letech k vývoji jazyka C.

Jedním z důvodů úspěchu a stálé síly jazyka C je, že ho programátoři mají rádi. Jazyk C spojuje obratnost a eleganci s hrubou silou a pružností. Je to strukturovaný jazyk, který neomezuje.

Jazyk C je důležitý i z jiného důvodu. Je branou ke dvěma dalším celosvětovým profesionálním programovacím jazykům: C++ a Java. C++ je vystavěn na C a Java je vystavěna na C++. C je také základem veškerého moderního programování a znalost C je základem pro úspěšné vytváření vysoce výkonného a kvalitního softwaru. [30]

11.2 Jazyk C++

Jazyk C++ je reakcí „céčkových“ programátorů na objektově orientované programování. Je vytvořen na pevných základech jazyka C a přináší podporu OOP (a mnoho dalších nových prvků).

11.2.1 Historie C++

C++ byl vytvořen Bjarne Stroustrupem v roce 1979 v Bellových laboratořích v New Jersey. Původně se jmenoval „C s třídami“. Jméno bylo změněno na C++ v roce 1983. Od té doby prošel jazyk C++ třemi zásadními revizemi. První z nich byla v roce 1985, druhá v roce 1990 a třetí proběhla během standardizačního procesu. Práce na standardizaci C++ započala před několika lety. Pro účely standardizace bylo vytvořeno spojení ANSI a ISO. Hrubý koncept plánovaného standardu byl vytvořen 25. ledna 1994. V tomto konceptu se komise ANSI/ISO C++ držela zásad definovaných Stroustrupem a přidala některé další. V podstatě však původní návrh reflektoval současný stav C++.

Brzy, po dokončení hrubého konceptu standardu, došlo k události, která způsobila, že standard byl podstatně rozšířen: bylo to vytvoření knihovny standardních šablon STL Alexandrem Stěpanovem.

Pravdou je, že standardizace C++ trvala mnohem déle, než kdokoliv očekával. Nicméně, hotový koncept prošel komisí úspěšně 14. listopadu 1997 a standard pro C++ se tak stal skutečností. [31]

11.2.2 Objektově orientované programování

Nabízí techniky, díky kterým lze zvládnout i velmi složité problémy. Účelem je dosáhnout opakované použitelnosti softwarových komponent a propojit data s úkoly, jež data zpracovávají.

Podstata OOP spočívá ve vymodelování „objektů“ (tedy reálných věcí) spíže než „dat“. Objekty, které modelujete, mohou být drobné prvky na obrazovce počítače jako tlačítka či okna se seznamem nabízených hodnot, nebo jimi mohou být reálné předměty jako kola, letadla, kočky či voda.

Objekty mají charakteristiky (rychlá, prostorná, černá, mokrá) a schopnosti (zrychlit, letět, přist, bublat). Úkolem OOP je tyto objekty vytvořit v programovacím jazyce.

C++ plně podporuje objektově orientované programování včetně jeho tří pilířů: zapouzdření, dědičnosti, polymorfismu. [32]

11.2.3 Zapouzdření

Je to mechanismus, který svazuje dohromady kód a data a zabezpečuje je před vnějšími zásahy či zneužitím. V objektově orientovaném jazyce může být kód s daty slučován takovým způsobem, že vznikají jakési nezávislé „černé skříňky“. Spojením kódu s daty, vzniká objekt. Jinými slovy lze říci, že objekt je instrument, který podporuje zapouzdření.

11.2.4 Dědičnost

Je to proces, při němž může jeden objekt získat vlastnosti jiného procesu. Přesněji může objekt zdědit obecnou sadu vlastností a do ní může přidat takové vlastnosti, které jsou specifické pouze pro něj. Dědičnost je důležitá, protože dovoluje objektu podporovat koncept hierarchické klasifikace.

11.2.5 Polymorfismus

Je to vlastnost, která umožňuje, aby bylo jediné jméno použito pro dva nebo více souvisejících ale technicky různých účelů. Ve vztahu k OOP dovoluje polymorfismus určit jedním jménem celou obecnou třídu procesů. Uvnitř obecné třídy procesů je pak volba konkrétního procesu dána typem dat. [31]

11.3 MULTIPLATFORMNÍ SOTWAROVÁ KNIHOVNA WXWIDGETS

WxWidgets (dříve wxWindows) je soubor softwarových knihoven vytvořených pro programovací jazyk C++ (volitelně Python, Lua, Perl), které dovolují kompilaci programů na několika různých softwarových platformách a překladačích s minimálními nutnými změnami zdrojového kódu vytvářené aplikace. Pro každou podporovanou platformu existuje jedna verze této knihovny. Prvotním cílem vývojářů bylo, aby aplikace vytvořené pomocí této knihovny byly k nerozeznání od aplikací vytvořených pomocí nativních překladačů daného operačního systému. Kromě základních API funkcí pro tvorbu GUI poskytuje knihovna další prvky vhodné pro přístup k ostatním, obecně dostupným technologiím cílového operačního systému. Knihovna navíc obsahuje další třídy, které nemalou měrou zjednodušují tvorbu programu a implementaci často používaných technologií a algoritmů. [29]

WxWidgets umožňuje vývojářům vytvářet aplikace pro operační systémy Win32, Mac OS X, GTL+, X11, Motif, WinCE a dalších za použití jednoho kódu. [33]

11.3.1 Historie wxWidgets

Vývoj této knihovny započal v roce 1992 student Julian Smart. Julian v té době vytvářel aplikaci, u které bylo požadováno, aby běžela jak na platformě Windows, tak v prostředí X-windows na Unixových pracovních stanicích. Všechny v té době dostupné komerční multiplatformní knihovny však byly příliš drahé a tak nezbývalo, než vytvořit knihovnu novou. Tak vznikla knihovna wxWindows, která je v současnosti šířena pod jménem wxWidgets. [29]

II. PRAKTICKÁ ČÁST

12 KONCEPCE TESTOVÁNÍ ŘADÍCÍCH ALGORITMŮ

Ještě před vlastním programováním bylo nutné vymyslet, jakým způsobem budu řadící algoritmy testovat, abych dostal co nejpřesnější výsledky odpovídající praktickému používání těchto algoritmů, a podle těchto nabytých poznatků se teprve pustit do samotného programování knihovny i vlastní testovací aplikace. Od začátku však již bylo zřejmé, že budu měřit rychlost, respektive čas řadícího algoritmu, nutný k setřídění posloupnosti prvků.

Po prohlédnutí několika odborných i méně odborných testů mi bylo jasné, že „běžné“ a velmi časté testování těchto algoritmů pouze na posloupnosti náhodných prvků, a to ještě typu kladné celé číslo, je v podstatě nic neříkající. Při praktickém použití řadících algoritmů v nejrůznějších programech se málokdy stane, že bychom řadili kompletně neseřazenou posloupnost prvků. Daleko běžnější je řazení například téměř seřazené posloupnosti. Stejně tak, ne vždy řadíme prvky datového typu integer. Každý algoritmus má také jiné vlastnosti, pracuje na jiných principech a ten algoritmus, který je rychlý na rozsáhlých posloupnostech prvků, může totálně „vyhořet“ na malých posloupnostech dat.

Rozhodl jsem se, že naprogramované řadící algoritmy budu testovat na generovaných, různě velkých posloupnostech prvků o datových typech integer, double a string a také na několika rozděleních testované posloupnosti, jinak řečeno „testovacích případech“, pro které naprogramuji konkrétní generátory. Testovací aplikace tedy musí umožňovat výběr datového typu, velikosti testované posloupnosti, testovacího případu, konkrétních algoritmů a samozřejmě i počtu opakování testu. Bylo by statisticky bezcenné, testovat každý algoritmus na každém testovacím případu pouze jednou. Test je nutné provést v rozsahu od deseti do tisíce opakování (pozn. statisticky by bylo určitě lepší provádět opakování testů v řádech tisíců, ale bohužel, a zvláště v případech rozsáhlých posloupností, je to z důsledku nutného času nemožné).

12.1 Testovací případy

Zvolil jsem pět testovacích případů (rozdělení generované posloupnosti prvků), na kterých budu data testovat.

1) Kompletně náhodné pole (angl. *completely random array*). Naivní lidé používají k testování pouze tento případ.

2) Již seřazené pole (angl. *already sorted array*). Tento případ je velmi důležitý, protože v praktickém používání se stává dosti často, že řadíme posloupnost pouze po minimálních modifikacích.

3) Již seřazené pole, ale v opačném pořadí (angl. *already sorted array in reverse order*). Mnoho algoritmů pracuje velmi pomalu na tomto druhu posloupnosti.

4) Již seřazené pole, ale posledních 256 prvků je náhodných (angl. *already sorted array where last 256 items are random*). Tímto testovacím případem simulujeme případ, kdy data uložená v datovém kontejneru jsou udržována seřazená a nové prvky přidáváme na konec.

5) Pole skládající se z identických prvků (Array consisting of identical elements). Mnoho algoritmů pracuje pomalu na takovém typu pole.

13 KNIHOVNA ŘADÍCÍCH ALGORITMŮ

13.1 Vývojové nástroje použité ke tvorbě knihovny

Knihovna *myStorage.cpp* byla vytvořena v programovacím jazyce C++ a ve vývojovém prostředí MS Visual Studio 2005 Professional. Toto vývojové prostředí jsem zvolil pro jeho stabilitu, rychlost, uživatelskou přívětivost a další profesionální vlastnosti.

13.2 Šablony

Hlavním požadavkem na knihovnu kromě toho, že má být založena na principech objektově orientovaného programování bylo to, že řadící metody musí být schopné řadit posloupnosti prvků všech běžných datových typů, včetně datového typu *string*. Z tohoto požadavku vyplývá použití šablon.

Pomocí šablon (v C++ klíčové slovo *template*) lze vytvářet tzv. generické funkce a třídy, ve kterých je datový typ zpracovávaných dat určen jako parametr. Jinými slovy řečeno, šablony umožňují elegantní vytvoření pouze jediné třídy pro všechny datové typy, místo pracného vytváření speciální třídy pro každý datový typ. Mnou vytvořená generická třída podporuje všechny výše zmíněné datové typy, a to bez nutnosti přetěžování funkcí, či operátorů (př. operátor ==). Pro datový typ *string* je použita knihovna <*string*>, ve které jsou již všechny potřebné operátory přetíženy. Pokud by však v budoucnu chtěl někdo knihovnu využít i například pro řazení objektů, tak by se nutnosti přetížení operátorů nevyhnul.

13.3 Třída myStorage

Srdcem celé knihovny je generická třída *myStorage*, která jak praví správné zásady objektově orientovaného programování, zapouzdřuje jednak atributy, tak i členské metody třídy. Pomocí tříd jazyce C++ vytváříme ve své podstatě nové datové typy, jejichž instance nazýváme objekty.

K atributům, tj. proměnným třídy umístěných v oblasti „private:“ lze mimo třídu přistupovat pouze voláním členských metod naší třídy, umístěných v oblasti „public:“ (uvnitř třídy lze přistupovat k atributům v „private:“ oblasti také pomocí metod v „private:“ oblasti.)

```
1  //deklarace šablony třídy myStorage
2  template <class T>
3  class myStorage
4  {
5   private:
6         int number, mnozina;
7         vector<T> myArray;
8         vector<T> mySortArray;
9   public:
10        myStorage (int number, int mnozina);
11        void Exchange(T& item1, T& item2);
12        /* ...
13        řadící metody
14        ...*/
15        int GetNumber(){return number;}
16        int GetMnozina(){return mnozina;}
17        void Vypis(); void Vypis2();
18        void GenerateInt();
19        void GenerateFloat();
20        void GenerateDouble();
21        void GenerateChar();
22        void GenerateString();
23 };
```

13.3.1 Atributy třídy myStorage

Generická třída myStorage obsahuje dvě proměnné typu integer, a to proměnnou *number*, ve které je uložen počet prvků posloupnosti a proměnnou *mnozina* jejíž číselná hodnota (1 - 5) určuje rozdělení (viz. podkapitola 12.1), se kterým generuji testovací posloupnost.

Pro uložení generované (*myArray*) a seřazené posloupnosti prvků (*mySortArray*) jsem použil datový kontejner z knihovny STL typu vector. Dá se říci, že vektor je dynamické jednorozměrné pole. Oproti klasickému poli má tu výhodu, že může svou velikost přizpůsobovat aktuálním požadavkům, měnících se v průběhu programu, tj. alokuje si paměť dle potřeby. Jinak se k vektoru můžeme chovat stejně, jako k normálnímu poli, což znamená možnost používat standardní indexování.

Metody, které jsem použil ve svém programu při práci s datovým kontejnerem vector:

- *vector::push_back* – Metoda přidává nový prvek na konec vektoru.

- *vector::size* – Metoda vrací počet prvků ve vektoru.
- *vector::clear* – Metoda odstraní všechny prvky vektoru.
- *vector::begin* – Metoda vrací iterátor odkazující na první prvek ve vektoru.
- *vector::end* – Metoda vrací iterátor odkazující na poslední prvek ve vektoru.

13.3.2 Konstruktor třídy myStorage

Třída má jeden parametrický konstruktor, jehož definice je následovná:

```
24 template <class T>
25 myStorage<T>::myStorage (int number, int mnozina)
26 {
27         this->number = number;
28         this->mnozina = mnozina;
29 }
```

Parametry konstruktoru slouží pouze pro předání hodnot atributům vytvářeného objektu.

```
30 myStorage<int> *storage = new myStorage<int>(50,1);
```

Př.: Dynamické vytvoření objektu *storage* šablony třídy myStorage, s datovým typem integer, počtem 50 prvků a s kompletně náhodným rozložením prvků.

13.3.3 Řadící metody třídy myStorage

Nejdůležitější částí třídy jsou samozřejmě naprogramované řadící algoritmy. Tyto však vzhledem k jejich množství v textu praktické části neuvedu, ale případné čtenáře odkážu na přílohu PI, kde jsou zdrojové kódy přiloženy, a také na popis jednotlivých algoritmů v teoretické části práce (viz. kapitola 4).

Některé algoritmy jsem naprogramoval ve dvou verzích. Jednak ručně, tj. napsáním vlastních funkcí a jednak za použití metod z knihovny <*algorithm*> z standardní knihovny šablon. Důvod, který mně k tomu vedl, byla možnost budoucího srovnání výkonu těchto verzí algoritmů.

Metody z knihovny <*algorithm*>, které jsem v programu použil:

- *std:make_heap* – vytvoření haldy (použito v algoritmu *StlHeapSort*)
- *std:sort_heap* – setřídění haldy (použito v algoritmu *StlHeapSort*)

- *std::inplace_merge* – setřídění dvou posloupností prvků (použito v algoritmu *StlMergeSort*)
- *std::min* – výběr minima ze dvou argumentů (použito v algoritmu *StlMergeSort*)
- *std::random_shuffle* – náhodné přeskupení prvků v zadaném rozsahu (použito v algoritmu *BogoSort*)

13.3.4 Metody třídy myStorage pro výpis prvků

Metody jsou ve třídě dvě, jedna pro výpis vygenerovaného vektoru *myArray* (*void Vypis()*) a druhá pro výpis seřazeného vektoru *mySortArray* (*void Vypis2()*).

Data vypisuji jak přes standardní datový proud *cout* na obrazovku, tak přes výstupní souborový proud *ofstream* do souboru.

```
31 ofstream outfile ("test.txt", ios_base::out);
```

Příklad vytvoření výstupního souborového proudu *outfile*. Soubor „*test.txt*“ se vytváří v adresáři, ve kterém se nachází zdrojové soubory knihovny.

```
32 cout<<myArray[i]<<" ";
33 outfile<<myArray[i]<<" ";
```

Do obou proudů (*cout* i *outfile*), poté zapisuji shodně za použití standardního operátoru <<.

13.3.5 Metody třídy myStorage pro generování posloupností prvků

S ohledem na jeden z požadavků testovaní algoritmů bylo potřeba vygenerovat posloupnosti všech datových typů o různých rozloženích (viz. podkapitola 12.1).

Za tímto účelem jsem vytvořil ve třídě myStorage jednu metodu pro každý datový typ (viz. řádky 18 - 22 v deklaraci třídy šablony myStorage). Tyto metody generují požadovanou posloupnost podle zadaných parametrů v konstruktoru při vytváření objektu.

Generování náhodných, respektive pseudonáhodných čísel se v jazyce C++ realizuje standardně pomocí funkcí *rand* a *srand*. Jestliže bych však v programu použil pouze funkci *rand*, byla by při opakovaném spuštění programu generována stále stejná posloupnost pseudonáhodných čísel. Vzhledem k tomuto faktu je nutností na začátku programu inicializovat tuto posloupnost pomocí již zmíněné funkce *srand*.

```
34 srand((unsigned) time(NULL));
```

Tímto příkazem na začátku funkce *int main ()* inicializuji generátor náhodných čísel.

K samotnému generování pseudonáhodných čísel pak používám funkci *rand*, která vrací "náhodná" čísla datového typu int v rozsahu 0 až RAND_MAX.

V případě, že chci vygenerovat číslo jiného datového typu než integer, např. typ double, je nutné návratovou hodnotu funkce *rand* přetypovat. Pokud chci vygenerovat číslo v intervalu <a;b>, tak je potřeba použít následující vzorec:

$$cislo = a + (b - a).(double)rand(\ ) / RAND\_MAX \tag{4}$$

Pro konkrétní popis implementace generátorů testovacích posloupností odkazuji na zdrojový kód knihovny, který se nachází v příloze PI.

14 TESTOVACÍ APLIKACE

Po vytvoření knihovny řadících algoritmů v jazyce C++ jsem se mohl pustit do tvorby aplikace, jež by mi umožnila naprogramované algoritmy otestovat. Jako prostředí pro tvorbu této aplikace jsem si vybral multiplatformní freewarovou knihovnu wxWidgets.

14.1 Vývojové nástroje použité ke tvorbě testovací aplikace

Aplikace *mySorting* byla vytvořena v programovacím jazyce C++ pomocí knihoven wxWidgets a ve vývojovém prostředí MS Visual Studio 2005 Professional s překladačem VC++. Jako distribuci wxWidgets jsem použil instalační balíček wxPack 2.8.7.03, který obsahuje hlavičkové a zdrojové soubory knihovny wxWidgets 2.8.7, application wizard a vizuální editor zdrojů pro wxWidgets wxFormBuilder 3.0.52.

14.2 Vytvoření kostry aplikace

Pomocí wxWidgets application wizardu jsem vytvořil kostru aplikace, která se skládala již z potřebných souborů pro spuštění aplikace. Jde o soubory mySorting.h a mySorting.cpp. V prvním z nich je odvozena vlastní třída myApp od základní třídy wxApp a přetížena virtuální funkce *wxApp::OnInit()*. Tato funkce je zavolána při inicializaci programu a zajišťuje úvodní operace programu.

Vzhledem k tomu, že pouze takhle jednoduché rozhraní by nebylo pro žádnou aplikaci dostačují, bylo nutné implementovat kvalitnější grafické uživatelské rozhraní.

14.3 Vytvoření grafického uživatelského rozhraní

Mé použité vývojové nástroje mi pro tvorbu GUI nabízely dvě možnosti. Buď všechny vizuální prvky naprogramovat ručně, a nebo použít součást balíčku wxPack, a to editor wxFormBuilder. Rozhodl jsem se pro druhou možnost. Nástroj wxFormBuilder je schopen sám generovat kód vytvářených prvků. K této činnosti využívá dvou souborů, v mém případě pojmenovaných mySorting_GUI.h a mySorting_GUI.cpp.

V prvním ze souborů odvozuje vlastní třídy od bázových tříd knihovny wxWidgets (např. třídu MainFrame od třídy wxFrame), kde atributy těchto tříd jsou použité ovládací prvky, které dané okno nebo dialog obsahuje. V souboru jsou samozřejmě pomocí makra *#include* zahrnuty všechny potřebné knihovny pro vizuální prvky. Ve druhém souboru se všechny

tyto prvky vytváří. Do zdrojového kódu umístěného v těchto dvou souborech, by se v žádném případě nemělo nijak zasahovat. Editor vytváří pro prvky typu okno nebo dialog nové soubory, kde odvozuje pro naše potřeby nové třídy, od ním již předtím vytvořených tříd, nacházejících se v souborech mySorting_GUI.h a mySorting_GUI.cpp (např. mySortingFrame od třídy MainFrame). Do těchto souborů již můžeme libovolně zasahovat a implementovat potřebné metody a obslužné rutiny událostí.

Pomocí wxFormBuilderu jsem tedy na již vytvořené okno vložil ovládací prvek panel (třída wxPanel), protože jsem se rozhodl na něj umisťovat všechny další ovládací prvky, nutné pro obsluhu mého programu. Pro umisťování ovládacích prvků na panel je vhodné použít speciálních objektů tzv. sizerů, které se starají o automatickou správu pozic a velikostí ovládacích prvků.

14.3.1 Rozmístění ovládacích prvků

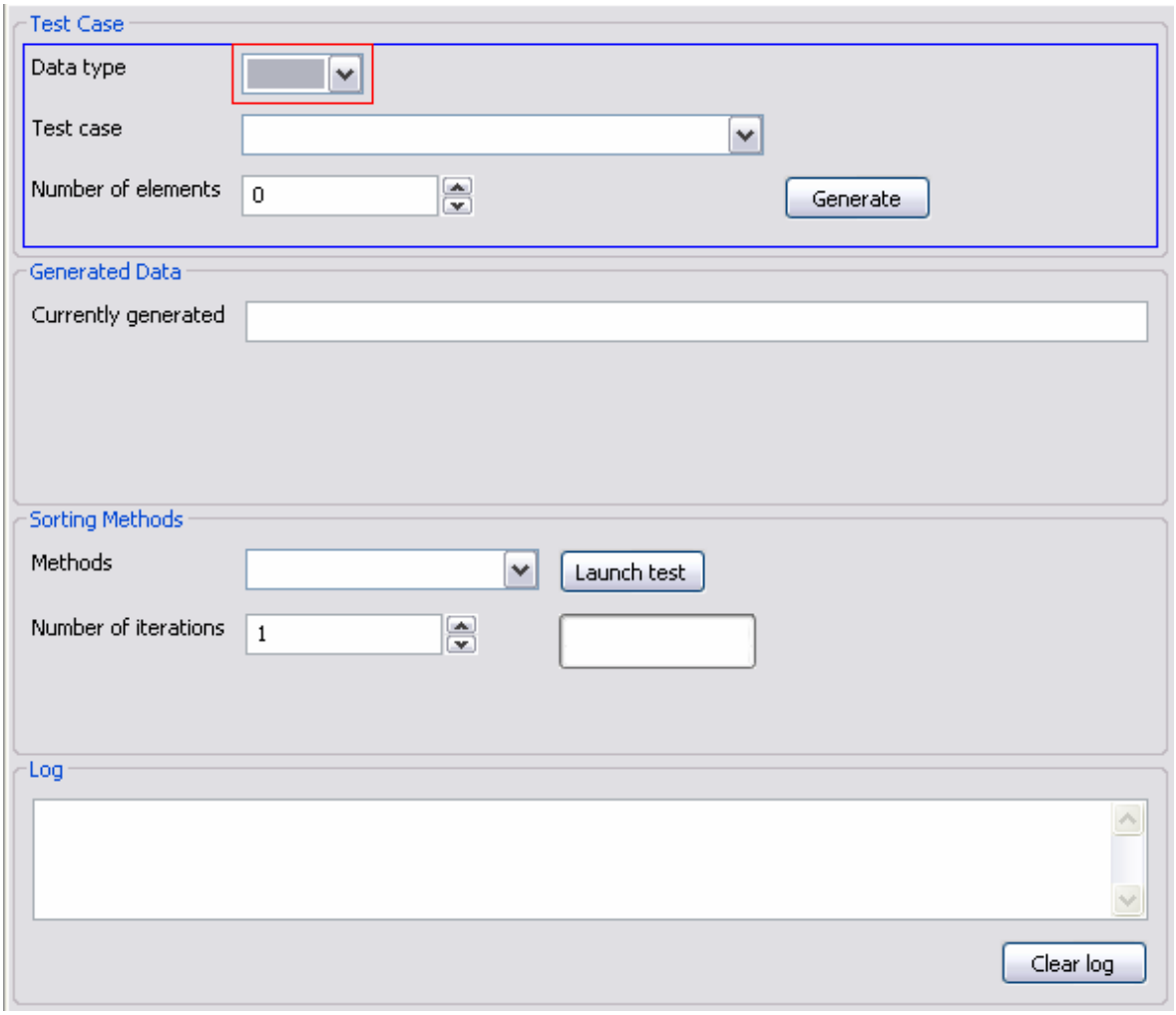
Na panel jsem umístil jeden hlavní sizer typu wxBoxSizer. Do tohoto sizeru jsem vložil další čtyři sizery typu wxStaticBoxSizer (podobný jako wxBoxSizer ale s parametrem label). Tímto se mi plocha panelu rozdělila na čtyři hlavní obdélníkové části, a to na sizery s labely Test case, Generated data, Sorting methods a Log.

Do sizeru s labelem *Test case* jsem vložil další sizer, ale typu wxFlexGridSizer. Tomuto sizeru jsem nastavil parametry 3 řádky a 3 sloupce. Do prvního sloupce jsem vložil prvky typu wxStaticText, které používám pro popis prvků ve druhém sloupci. Do toho jsem vložil dva comboboxy typu wxComboBox, sloužících k výběru datového typu a testovacího případu, a prvek typu wxSpinCtrl, který slouží k výběru počtu prvků. Ve třetím sloupci je pouze tlačítko typu wxButton s labelem *Generate*.

Do sizeru s labelem *Generated data* jsem vložil opět popisek typu wxStaticText a objekt typu wxTextCtrl, který slouží pro výpis vygenerované posloupnosti.

Do sizeru s labelem *Sorting methods* jsem umístil další sizer typu wxFlexGridSizer o dvou řádcích a třech sloupcích. V prvním sloupci jsou opět popisky pro prvky nacházející se druhém sloupci, a to combobox pro výběr řadícího algoritmu a wxSpinCtrl, umožňující výběr počtu opakování testu. Ve třetím sloupci se nachází tlačítko s labelem *Launch test* a pod ním je umístěn objekt typu wxGauge, umožňující jednoduché znázornění průběhu testu.

Do posledního obdélníkového sizeru s labelem *Log* je umístěn prvek typu wxTextCtrl, který slouží k výpisu výsledků testu a tlačítko s labelem *Clear log*, umožňující smazání logu.



Obrázek 4: Rozmístění ovládacích prvků v editoru wxFormBuilder

Pomocí editoru jsem vytvořil také dialog typu wxDialog, pomocí kterého je zpřístupněna nápověda. a stručné informace o programu. Tímto má práce s editorem wxFormBuilder skončila. Další částí práce byla implementace již naprogramované knihovny myStorage do testovací aplikace a naprogramování všech obslužných rutin pro ovládací prvky.

14.4 Popis implementace knihovny myStorage do testovací aplikace

Vzhledem ke skutečnosti, že knihovna wxWidgets je plně vytvořena pomocí objektově orientovaného jazyka C++, nebyla implementace mnou vytvořené knihovny myStorage v jazyce C++ do testovací aplikace mySorting úkolem nikterak obtížným.

Knihovnu *myStorage.cpp* bylo nejprve nutné přidat v Solution exploreru v prostředí MS Visual Studia do projektu mySorting mezi zdrojové soubory projektu.

Dalším krokem bylo vložení knihovny myStorage pomocí direktivy preprocesoru *#include "myStorage.cpp"* do některého z hlavičkových souborů projektu. Já jsem toto makro umístil do hlavičkového souboru mySortingPanel.h, protože právě v deklaraci třídy mySortingPanel vytvářím ukazatele všech požadovaných datových typů na typ šablony třídy myStorage.

Další práce s knihovnou v prostředí wxWidgets je naprosto identická, jako v jakémkoliv jiném prostředí založeném na jazyce C++. Objekty se vytváří stejným způsobem, a totéž se dá říci i o volání konkrétních metod třídy.

Co je potřeba si uvědomit je to, že přístup k atributům třídy myStorage mají podle zásad OOP přístup jen členské metody této třídy. K atributům number a množina tedy přistupuji, přes již naprogramované funkce *int GetNumber()*, *int GetMnozina()*.

14.5 Popis aplikace

Program *mySorting* je naprogramován na míru požadavkům, které jsem si stanovil při vymýšlení koncepce testování řadících algoritmů (viz. kapitola 12). V části okna nazvané *Test case* si můžu libovolně navolit požadovaný testovací případ. Vybírám datový typ, rozdělení testovací posloupnosti (viz podkapitola 12.1) a počet prvků posloupnosti. Stisknutím tlačítka Generate s identifikátorem ID_CLICK_1 se vyvolá událost zmáčknutí tlačítka wxEVT_COMMAND_BUTTON_CLICKED, na kterou je v tabulce událostí nacházející se uvnitř souboru mySortingPanel.cpp, namapována makrem EVT_BUTTON(ID_CLICK_1, mySortingPanel::OnClick1) obslužná rutina *void OnClick1(wxCommandEvent& event),* jež je členskou metodou třídy mySortingPanel.

V této rutině vybírám hodnoty z comboboxů pomocí funkce *wxComboBox::GetValue* a získané hodnoty si ukládám do členských proměnných třídy mySortingPanel ve formátu wxString. Následně podle získaných parametrů vytvářím (generuji) správný typ objektu typu šablony třídy myStorage a vypisuji momentálně vygenerovanou posloupnost prvků pomocí členské metody *void mySortingPanel::vypisDoTextCtrl1()* do prvku m_textCtrl1 nacházejícího se v oddílu oknu nazvaném *Generated data*.

```
35 if(m_comboBox1->GetValue()==wxT("int"))
36 {
37  if(m_comboBox2->GetValue() == wxT("Completely random array"))
38  {
39        vyberMnoziny = 1; //mnozina
40  }
41  skladInt = new myStorage<int>(m_spinCtrl1-GetValue(),vyberMnoziny);
42  skladInt->GenerateInt();
43 }
```

Př.: Dynamické vytvoření objektu skladInt typu šablony třídy myStorage, s datovým typem integer a parametry (hodnota získaná z wxSpinCtrl, vyberMnoziny = hodnota nastavená v rozmezí 1-5 podle hodnoty m_comboBox2).

Když mám vygenerovaná data, tak můžu přejít ke druhé části výběru parametrů testu, a ta se nachází v oddíle okna nazvaném *Sorting methods*. V comboboxu nazvaném *Methods* vybírám požadovanou metodu, kterou jsem se rozhodl otestovat. V prvku wxSpinCtrl poté vybírám počet opakování testu, a to z důvodu, že by bylo nesmyslné testovat algoritmus na vygenerovaných datech pouze jednou. Pro kvalitní statistické informace potřebujeme opakování více. Program funguje tím způsobem, že pokud necháme hodnotu v wxSpinCtrl nastavenou na 1, tak můžeme na stále stejné vygenerované posloupnosti testovat více algoritmů. Pokud však nastavíme hodnotu wxSpinCtrl > 1, po každém opakování testu se pro vybraný algoritmus znovu zavolá funkce *GenerateInt()* (nebo jiná funkce na generování prvků, záleží na datovém typu) a vygenerují se nová data. Je to z toho důvodu, že 100 opakování např. Bubble Sortu na stále stejné posloupnosti 1000 náhodně vygenerovaných prvků, nám řekne mnohem méně, než když stejný algoritmus testujeme stokrát na různých posloupnostech 1000 náhodně vygenerovaných prvků. Nevytváří se tedy znovu celý nový objekt (všechny parametry navolené v oblasti Test Case zůstávají stejné) ale jen se generuje nová posloupnost.

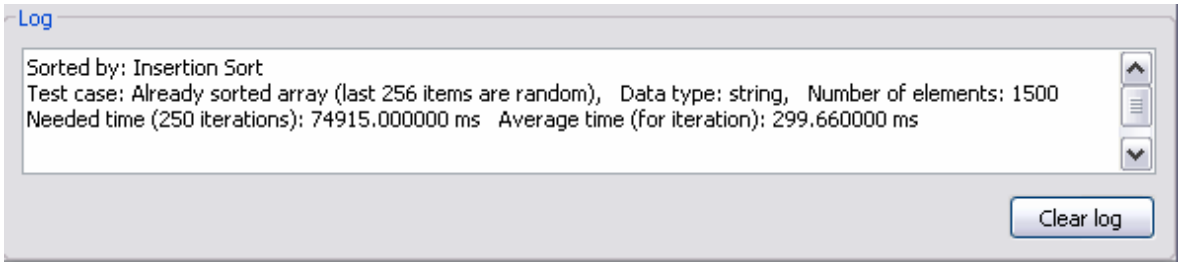
Co se tedy odehrává v programu, pokud zmáčknu tlačítko *Launch test*? Zmáčknutí tlačítka opět vyvolá událost wxEVT_COMMAND_BUTTON_CLICKED, na kterou je v tabulce událostí namapována makrem EVT_BUTTON(ID_CLICK_2, mySortingPanel::OnClick2) obslužná rutina *void OnClick2(wxCommandEvent& event)*. V této rutině pomocí metody *wxComboBox::GetValue* vybírám hodnoty z comboboxu a wxSpinCtrl. Podle těchto hodnot volám požadovanou řadící metodu, měřím čas metody pomocí objektu wxStopWatch, seřazenou a neseřazenou posloupnost *vypisuji do souboru* „*test.txt*“,

nacházející se ve stejném adresáři, jako spustitelný soubor aplikace mySorting. Výsledky vypisuji do prvku wxTextCtrl s názvem Log a nastavuji parametry objektu wxGauge, který jednoduchým způsobem zobrazuje průběh testu.

```
44 pocetOpakovani = m_spinCtrl2->GetValue();
45  if(genDatTyp == wxT("int"))
46  {
47        if(m_comboBox3->GetValue() == wxT("Bubble Sort"))
48        {
49              usedMetoda = wxT("Bubble Sort");
50              m_gauge1->SetRange(pocetOpakovani);
51              for(int i = 1;i <= pocetOpakovani; i++)
52              {
53                    skladInt->Vypis();
54                    wxStopWatch sw;
55                    skladInt->BubbleSort();
56                    sw.Pause();
57                    skladInt->Vypis2();
58                    if(pocetOpakovani > 1)
59                          skladInt->GenerateInt();
60                    celkovyCas += sw.Time();
61                    m_gauge1->SetValue(i);
62              }
63              vypisDoTextCtrl2();
64        }
```

Př.: Ukázka aplikace metody Bubble Sort na předem vygenerovanou posloupnost prvků skrývající se v objektu typu myStorage *skladInt*.

Metoda *vypisDoTextCtrl2()*, slouží k výpisu do prvku m_textCtrl2, který slouží k „logování“ výsledků testů. Výpis vypadá tak, že v prvním řádku je vypsána testovaná řadící metoda. Ve druhém řádku je vypsáno v jakém rozložení byla naše posloupnost vygenerována, datový typ a počet prvků posloupnosti. Ve třetím řádku je vypsán celkový čas, nutný k seřazení posloupnosti v milisekundách za všechny opakování a průměrný čas seřazení posloupnosti v milisekundách za jedno opakování. Je také nutno dodat, že obsah logu vymažeme stisknutím tlačítka *Clear log*.



Obrázek 5: Výpis do Logu v aplikaci mySorting

Okno obsahuje také jednoduchý menubar, kde v menu *Help* stisknutím tlačítka *About* (klávesová zkratka F1) zobrazíme objekt typu wxDialog, na kterém je umístěna nápověda programu.

15 TESTOVÁNÍ ŘADÍCÍCH ALGORITMŮ

Po naprogramování knihovny algoritmů i vlastní testovací aplikace mySorting jsem mohl postoupit k testování jednotlivých algoritmů.

Testoval jsem datové typy integer, double a string (řetězec je složen z 5 znaků v rozsahu písmen a-z) na posloupnostech o 100, 5.000 a 100.000 prvků, abych simuloval chování algoritmů na krátkých, středních a rozsáhlých posloupnostech prvků.

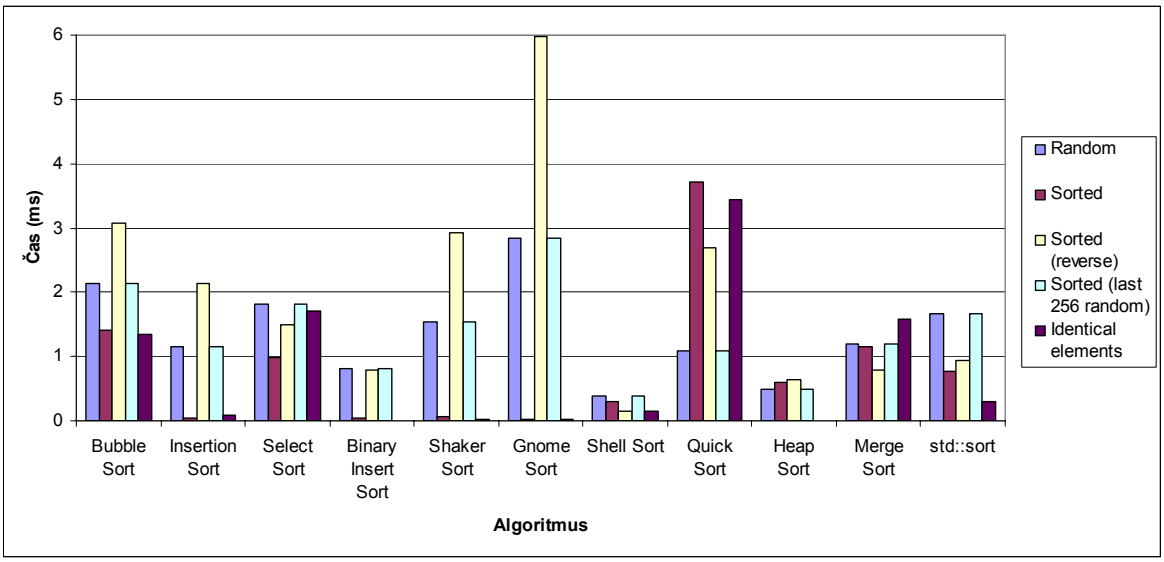
Do testu jsem zapojil 6 algoritmů s asymptotickou časovou složitostí $O\left(n^2\right)$, 3 algoritmy s časovou složitostí $O\left(n\log n\right)$ (pozn.: algoritmus Shell Sort má ve skutečnosti složitost $O\left(n\log^2 n\right)$, ale v testu jsem jej zařadil mezi algoritmy této kategorie), a také pro srovnání algoritmus ze standardní knihovny šablon jazyka C++, který jsem v testu pojmenoval std::sort. V praktické části práce uvádím z důvodu přehlednosti jako výsledky pouze grafy. Přesné časy, které jsou uvedené ve formě tabulek, jsou v příloze PII. Každý sloupec v grafu vyjadřuje průměrný čas v milisekundách, který algoritmus potřeboval k jednomu seřazení vygenerované posloupnosti prvků.

15.1 Testovací podmínky

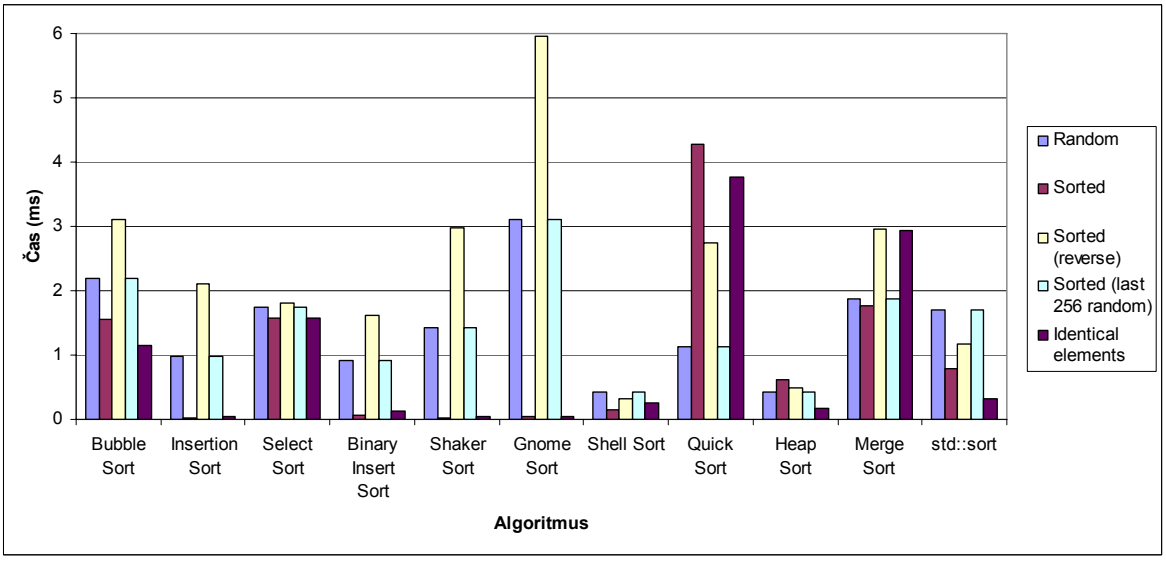
Testoval sem na počítači s následujícím vybavením:

- Operační systém Microsoft Windows XP Professional SP2
- Procesor Intel Core 2 Duo E4300, 1800 MHz
- Chipset Intel Broadwater P965
- Pracovní paměť 2048 MB (DDR2-800 DDR2 SDRAM)

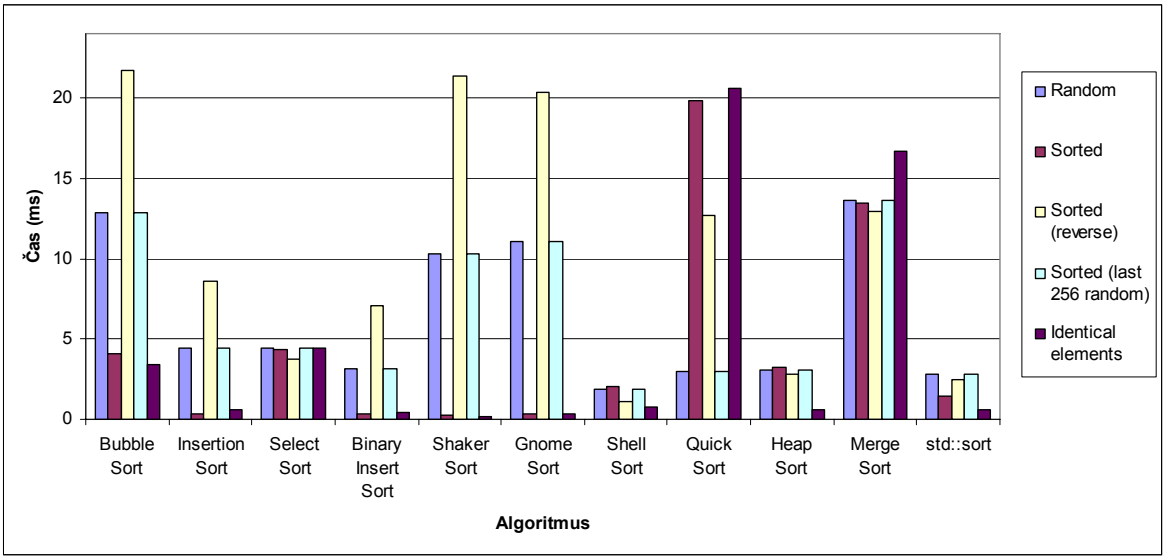
15.2 Řazení posloupnosti 100 prvků



Graf 1: Rychlost seřazení posloupnosti 100 prvků datového typu integer .



Graf 2: Rychlost seřazení posloupnosti 100 prvků datového typu double.



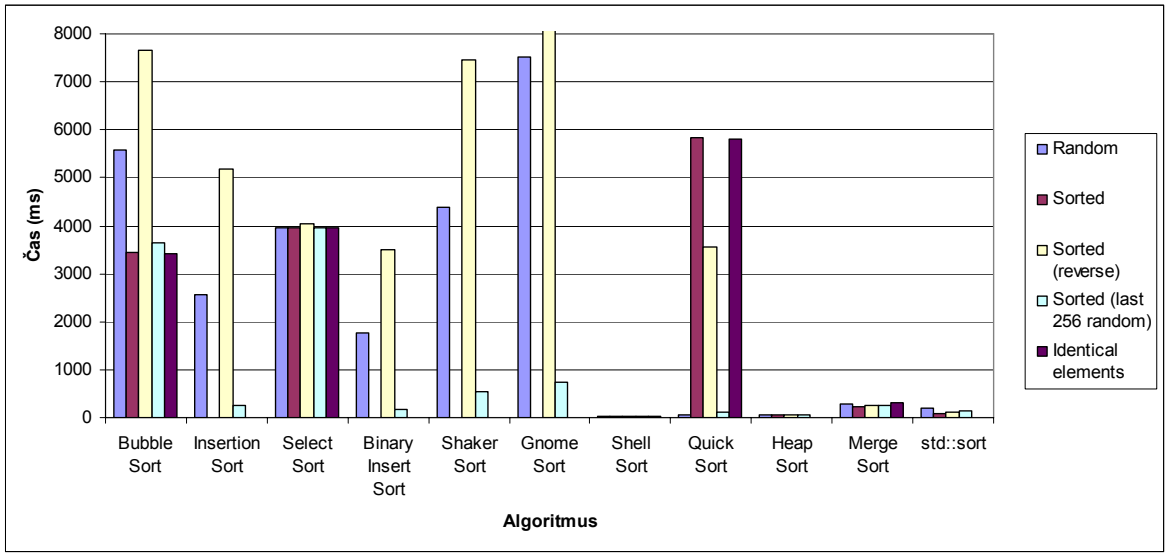
Graf 3: Rychlost seřazení posloupnosti 100 prvků datového typu string.

Prvním testem bylo vyzkoušet chování testovaných algoritmů na krátké posloupnosti prvků, konkrétně na posloupnosti o 100 prvcích. Testovací případ „kompletně náhodné pole“ je v tomhle případě kvůli počtu prvků identický s případem „seřazené pole (posledních 256 prvků je náhodných)“.

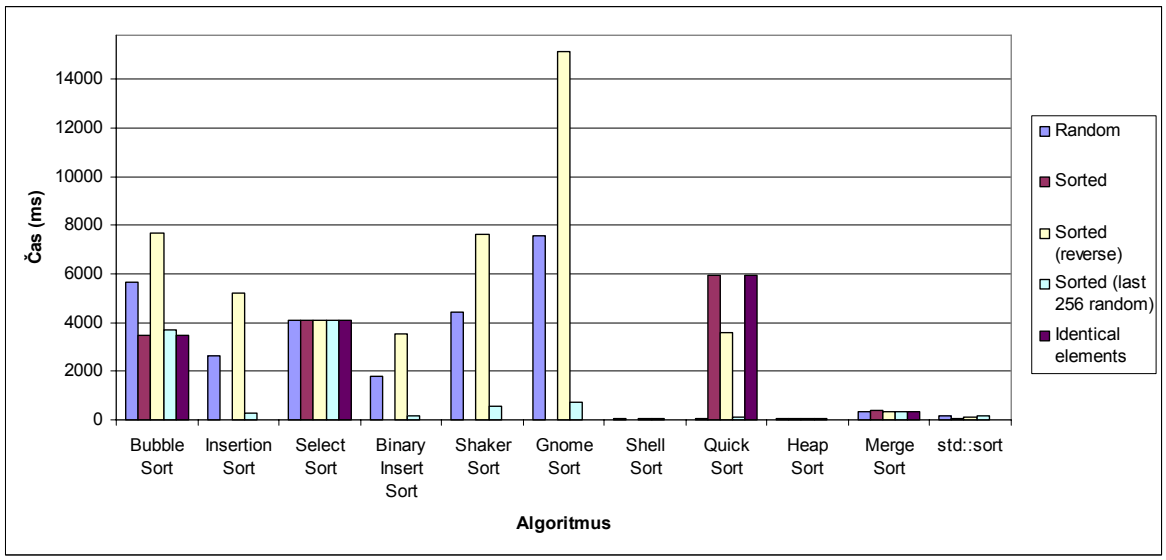
Algoritmus Bubble Sort nepotvrdil teoretický poznatek, že řadí ze všech algoritmů s časovou složitostí $O(n^2)$ nejrychleji již seřazené pole prvků. Naopak, dosahoval na tomto druhu posloupnosti, spíše podprůměrných výsledků ve srovnání s jeho variantami Shaker Sort a Gnome Sort. Všechny tyto tři zmíněné algoritmy také dosáhly velmi špatného výsledku při řazení seřazeného pole v opačném pořadí. Velmi dobrých výsledků se oproti tomu dopracovaly obě verze Insertion Sortu, a to především při řazení již seřazené posloupnosti a posloupnosti skládající se z identických prvků, čímž se potvrdily teoretické poznatky. Selection sort se ukázal být algoritmem velmi univerzálním, protože nabýval velmi podobných časů na všech rozděleních posloupnosti. Mezi algoritmy s časovou složitostí $O(n \log n)$ mě mírně překvapilo jen horší chování Quick Sortu na jiné, než kompletně náhodné posloupnosti. Na kompletně seřazeném poli a poli skládajícím se z identických prvků dosahoval nejhorších výsledků. Pomalým se ukázal být pro tento počet prvků také algoritmus Merge Sort. Největší kvality v tomto testu ukázal algoritmus Shell Sort, který nabýval nejlepších časů u všech testovaných datových typů.

Tento test prokázal, že pro malý počet prvků se vyplatí použití algoritmů s asymptotickou časovou složitostí $O(n^2)$, především tedy Selection Sortu, nebo jedné z verzí Insertion Sortu. Implementace těchto algoritmů je snadnější, než implementace některého ze složitějších algoritmů a rychlostní výsledky jsou srovnatelné.

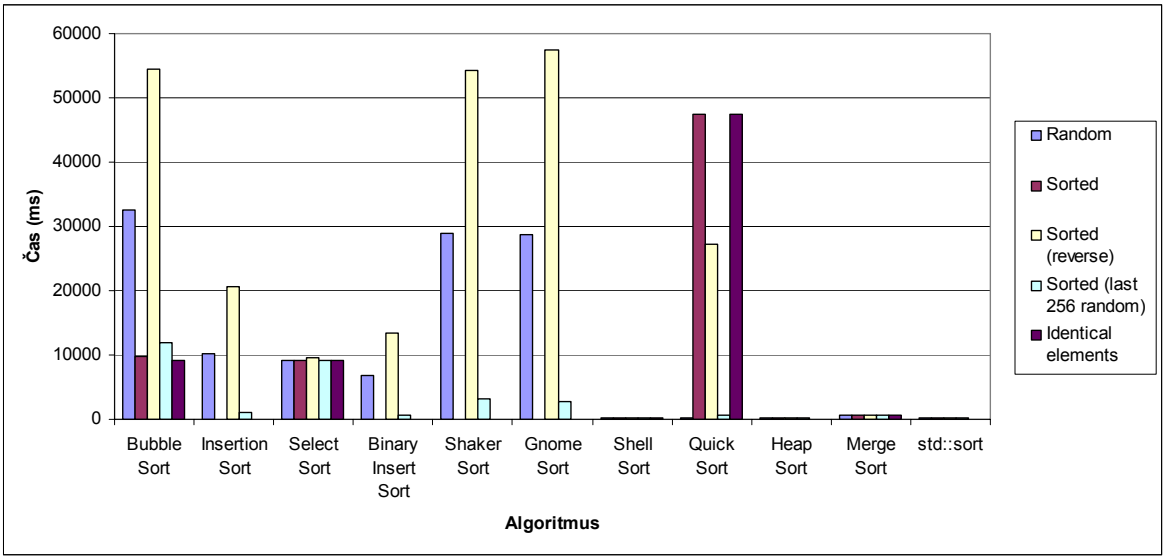
15.3 Řazení posloupnosti 5.000 prvků



Graf 4: Rychlost řazení posloupnosti 5.000 prvků datového typu integer.



Graf 5: Rychlost seřazení posloupnosti 5.000 prvků datového typu double.

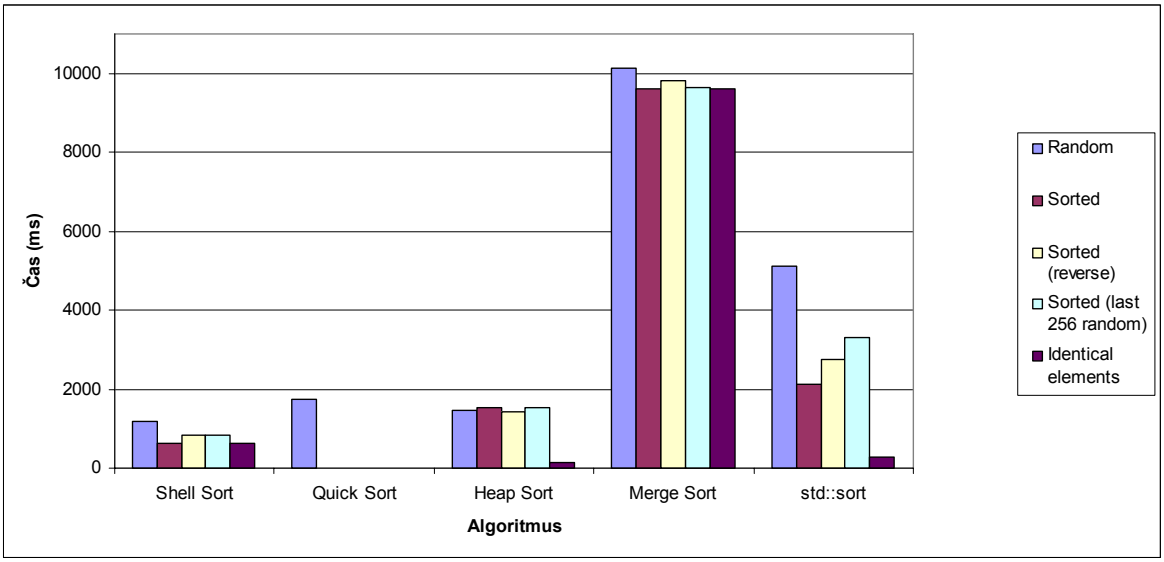


Graf 6: Rychlost seřazení posloupnosti 5.000 prvků datového typu string.

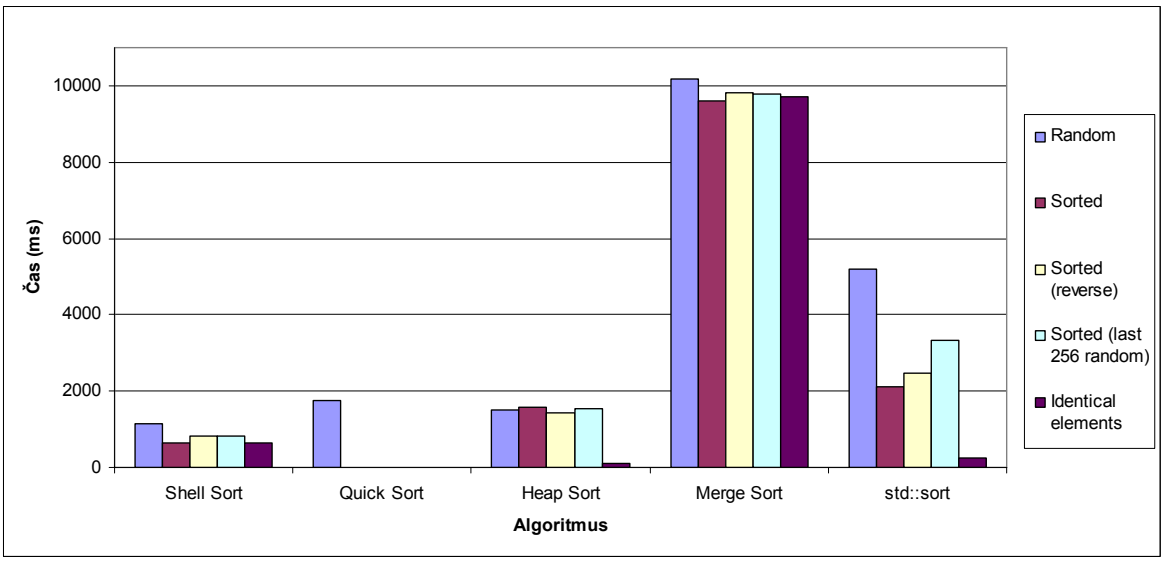
Druhým testem bylo vyzkoušet chování algoritmů na středně rozsáhlé posloupnosti prvků, konkrétně na posloupnosti o 5.000 prvcích. U takového počtu prvků se už naplno ukazuje rozdíl mezi algoritmy s časovou složitostí $O(n^2)$, které dosahují znatelně horších výsledků, než algoritmy se složitostí $O(n \log n)$. Bubble Sort, stejně jako Selection Sort se ukázaly být velmi pomalými algoritmy pro takovýto počet prvků. Selection Sort ale oproti Bubble Sortu zase prokázal svoji univerzalitu a dosahoval velmi podobných časů na všech rozděleních posloupnosti. Algoritmy Shaker Sort a Gnome Sort nabývaly v podstatě velmi podobných výsledků, kdy byly nejrychlejší na již seřazeném poli a na poli skládajícím se z identických prvků. Jejich výsledky na kompletně náhodném, a především seřazeném poli v opačném pořadí patřili k nejhorším, proto jejich používání nemohu doporučit. Obě varianty Insertion Sortu se znovu ukázaly jako velmi efektivní a podle výsledků je zřejmé, že z algoritmů se složitostí $O(n^2)$, byly za oba testy nejvýkonnější. U algoritmů s časovou složitostí $O(n \log n)$ byl nejrychlejším Shell Sort, následován Heap Sortem a std::sortem. Algoritmus Merge Sort oproti těmto třem algoritmům opětovně zaostával. Samostatnou kapitolou je zase Quick Sort, který ač jeden z nejrychlejších při řazení kompletně náhodného pole, pak výrazně zaostává na dalších testovacích případech. Výjimkou je případ seřazeného pole, kdy posledních 256 prvků je náhodných. Vysvětlit se to dá tím, že implementace Quick Sortu, kterou jsem použil, používá pevný výběr pivota, a to poslední prvek posloupnosti. Pokud by se použil nějaký optimalizovanější postup výběru pivota,

jeho výkon by byl vyšší. Lepší srovnání algoritmů s časovou složitostí $O(n \log n)$ přinese až poslední test.

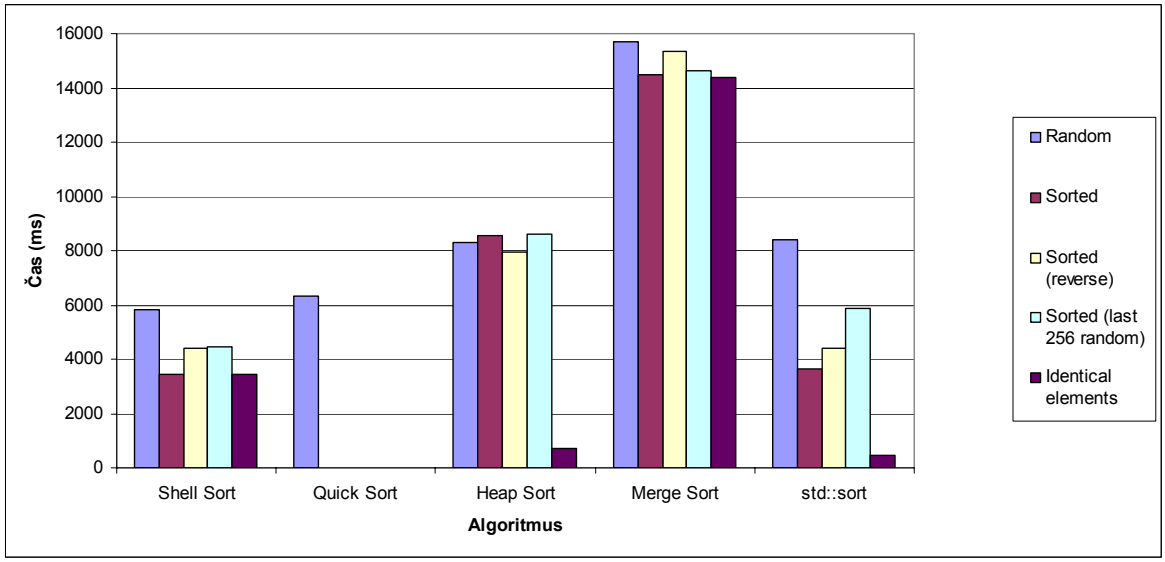
15.4 Řazení posloupnosti 100.000 prvků



Graf 7: Rychlost řazení posloupnosti 100.000 prvků datového typu integer.



Graf 8: Rychlost seřazení posloupnosti 100.000 prvků datového typu double.



Graf 9: Rychlost seřazení posloupnosti 100.000 prvků datového typu string.

Posledním testem bylo vyzkoušet chování algoritmů na rozsáhlé posloupnosti prvků, konkrétně na posloupnosti o 100.000 prvcích. V tomto testu už nefigurují algoritmy s časovou složitostí $O\left(n^2\right)$, protože jimi dosažené časy by byly nyní již nesrovnatelné. Merge Sort se znovu projevil jako nejpomalejší algoritmus patřící do této kategorie, čímž se potvrdily teoretické poznatky, že z algoritmů této kategorie je nejpomalejší. U Quick Sortu jsem musel přeskočit testování na jiné než náhodné posloupnosti, protože jím dosažené časy byly příliš vysoké a výrazně by pokazily měřítko v grafu. Vysvětlení špatného výkonu Quick Sortu jsem poskytl již v předchozím testu. V tomto testu se nejlépe chovaly algoritmy Shell Sort a Heap Sort u datových typů integer a double. U řetězců to byl Shell Sort a std::sort.

Největší kvality ve všech třech testech ukázal algoritmus Shell Sort. Tento algoritmus je nejenom jednoduchý na implementaci ale dosahoval velmi vyrovnaných časů u všech testovacích případů. Jeho výkon je srovnatelný i s knihovním algoritmem std:sort, který je kombinací několika řadících algoritmů a je optimalizován pro různé rozložení testované posloupnosti.

ZÁVĚR

Ze začátku teoretické části mé diplomové práce jsem se zaměřil na obecný popis algoritmů a vysvětlení důležitých pojmů, které s algoritmy souvisí. Soustředil jsem se především na možnosti porovnávání a analýzu výkonnosti algoritmů. Další část mé práce je již čistě cílena na rozsáhlou problematiku řazení a řadících algoritmů. V práci se snažím obsáhnout co nejvíce známých řadících metod, jejich variant a popsat rozdíly v jejich fungování a efektivitě řazení dat. V průběhu kapitoly o řazení se zaměřuji především na popis principů, srovnání a analýzu vnitřních metod řazení, ale v závěru kapitoly jsou stručně uvedeny i principy vnějších řadících metod.

V praktické části jsem vytvořil programovou knihovnu řadících algoritmů založenou na objektově orientovaném programování. Hlavním požadavkem na knihovnu bylo, že řadící metody v ní obsažené, musí být schopny řadit posloupnosti prvků pro libovolný běžný datový typ. Základem knihovny je proto generická třída, která zapouzdřuje všechny řadící metody a datový kontejner vektor.

V další části práce jsem naprogramoval testovací aplikaci v prostředí wxWidgets, ve které je knihovna obsahující řadící metody implementována. Pomocí této aplikace jsem byl schopen řadící algoritmy testovat a porovnat jejich výkon s teoretickými poznatky. Algoritmy testuji na pěti rozdílných rozloženích posloupnosti prvků, různých datových typech a s rozdílným počtem prvků v posloupnosti.

Výsledky testů bych shrnul následovně. Ukázalo se, že není příliš velký rozdíl ve výkonu algoritmů na různých datových typech. Samozřejmě, že s přibývající délkou řetězce, by klesal výkon algoritmů na datovém typu string. Na krátkých posloupnostech prvků se příliš nevyplatí používat komplikovanější metody s časovou složitostí $O(n \log n)$ ale bohatě postačí použít některou z jednodušších metod, například metodu Binary Insertion Sort. Tato metoda se mi jevila v průběhu celého testování jako ta nejefektivnější z metod s časovou složitostí $O(n^2)$. Na rozsáhlejších posloupnostech prvků už je situace úplně jiná. Použití jednodušších metod na posloupnosti o několika tisících prvků nemohu doporučit. Výjimkou potvrzující pravidlo byl algoritmus Shell Sort. Tento algoritmus je relativně jednoduchý na implementaci a přesto velmi výkonný. Dobrou alternativou k ručně napsaným algoritmům je algoritmus Sort z knihovny STL, který podával velmi stabilní výsledky na všech typech posloupností.

SEZNAM POUŽITÉ LITERATURY

[1] BAYER, Tomáš. Programování I. [online]. 2007 [cit. 2008-04-12]. Dostupný z WWW: <http://www.natur.cuni.cz/~bayertom/Prog1/programovani2.pdf>.

[2] KRÁTKÝ, Michal. Úvod do programování [online]. 2004 [cit. 2008-04-12]. Dostupný z WWW:

<http://www.cs.vsb.cz/kratky/courses/2004-05/udp/presentation/udp-1_6.pdf>.

[3] Algoritmus [online]. 2006 [cit. 2008-04-12]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Algoritmus>.

[4] WROBLEWSKI, Piotr. Algoritmy - datové struktury a programovací techniky. 1. vyd. Brno : Computer Press, 2004. 351 s. ISBN 80-251-0343-9.

[5] KRESLÍKOVÁ, MARTINEK. Složitost algoritmů [online]. 2004 [cit. 2008-04-14]. Dostupný z WWW: <http://jaja.kn.vutbr.cz/~belovic/IZP/8c_slozitost.pdf>.

[6] ČERNÝ , Jakub. Jak porovnávat algoritmy? [online]. 2008 [cit. 2008-04-13]. Dostupný z WWW: <http://kam.mff.cuni.cz/~kuba/ka/cas_sloz.pdf>.

[7] Řadící algoritmus [online]. 2006 , 26. 2. 2008 [cit. 2008-04-19]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Řadicí_algoritmus>.

[8] PINDUR, Petr. Řazení [online]. 1999 [cit. 2008-04-18]. Dostupný z WWW: <www.penguin.cz/~pindur/inf/ADS3.doc>.

[9] Počítače a programování [online]. 2004 [cit. 2008-04-19]. Dostupný z WWW: <http://www.kiv.zcu.cz/~kleckova/Predmety/PPA1/download/PPA1_9_2005.pdf>

[10] BAYER, Tomáš. Třídící algoritmy [online]. 2007 [cit. 2008-05-01]. Dostupný z WWW: <http://www.natur.cuni.cz/~bayertom/Prog1/prog5.pdf>.

[11] Selection sort [online]. 2002 , 28 April 2008 [cit. 2008-05-01]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Selection_sort>.

[12] Selection Sort [online]. 2005 [cit. 2008-05-01]. Dostupný z WWW: <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm>.

[13] Bubble sort [online]. 2002 , 2 May 2008 [cit. 2008-05-01]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Bubble_sort>.

[14] Insertion sort [online]. 2008 [cit. 2008-05-03]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Insertion_sort>.

[15] Tree sort [online]. 2006 [cit. 2008-05-03]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Binary_tree_sort>.

[16] Library sort [online]. 2005 [cit. 2008-05-03]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Library_sort>.

[17] Sorting algorithm [online]. 2001 , 26 April 2008 [cit. 2008-05-03]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Sorting_algorithm>.

[18] Shell Sort [online]. 2005 [cit. 2008-05-03]. Dostupný z WWW: <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/shellSort.htm>.

[19] Gnome sort [online]. 2004 , 21 April 2008 [cit. 2008-05-04]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Gnome_sort>.

[20] Gnome sort [online]. 2007 [cit. 2008-05-04]. Dostupný z WWW: <http://www.nist.gov/dads/HTML/gnomeSort.html>.

[21] Úvod do programování [online]. 2005 [cit. 2008-05-04]. Dostupný z WWW: <http://www.cs.vsb.cz/kratky/courses/2004-05/udp/presentation/udp-6_6.pdf>.

[22] VOLF , Přemysl. Třídící algoritmy [online]. 2004 , 11.1.2007 [cit. 2008-05-04]. Dostupný z WWW: <http://agents.felk.cvut.cz/teaching/x33dsp/DSP4%20-%20tridici%20algoritmy.pdf>.

[23] Heapsort [online]. 2001 , 29 April 2008 [cit. 2008-05-04]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Heapsort>.

[24] Quicksort [online]. 2006 , 30. 3. 2008 [cit. 2008-05-04]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Quicksort>.

[25] Merge sort [online]. 2006 , 4. 5. 2008 [cit. 2008-05-05]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Merge_sort>.

[26] Counting sort [online]. 2002 , 30 April 2008 [cit. 2008-05-05]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Counting_sort>.

[27] Bucket sort [online]. 2002 , 30 April 2008 [cit. 2008-05-05]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Bucket_sort>.

[28] Radix sort [online]. 2001 , 10 April 2008 [cit. 2008-05-05]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Radix_sort>.

[29] BLIŽňÁK, Michal. Systémové programování. Zlín : Univerzita Tomáše Bati, 2005. 202 s. ISBN 80-7318-364-1.

[30] SCHILDT, Herbert. Nauč se sám C. Praha : SoftPress, 2001. 624 s. ISBN 80-86497-16-X

[31] SCHILDT, Herbert. Nauč se sám C++. Praha : SoftPress, 2001. 624 s. ISBN 80-86497-13-6.

[32] LIBERTY, Jesse. Naučte se C++ za 21 dní. Brno : Computer Press, 2007. 796 s. ISBN 978-80-251-1583-1.

[33] WxWidgets [online]. 2002 [cit. 2008-01-23]. Dostupný z WWW: http://www.wxwidgets.org/.

[34] Bogosort [online]. 2002 , 26 April 2008 [cit. 2008-05-05]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Bogosort>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ANSI American National Standards Institute.

API Application Programming Interface

BCPL Basic Combined Programming Language

ČSN Česká technická norma.

GNU GNU's Not Unix.

GUI Graphical User Interface

ISO International Standards Organization.

LSD Least Significant Digit.

MFC Microsoft Foundation Class Library

MSD Most Significant Digit.

OOP Objektově orientované programování.

XML eXtensible Markup Language

SEZNAM OBRÁZKŮ

Obrázek 1: Příklad haldy 27
Obrázek 2: Rozdělení posloupnosti vzhledem k pivotu $\overline{X}$ 37
Obrázek 3: Schéma postupu třípáskových metod 47
Obrázek 4: Rozmístění ovládacích prvků v editoru wxFormBuilder 62
Obrázek 5: Výpis do Logu v aplikaci mySorting 66

SEZNAM GRAFŮ

Graf 1: Rychlost seřazení posloupnosti 100 prvků datového typu integer 68
Graf 2: Rychlost seřazení posloupnosti 100 prvků datového typu double. 68
Graf 3: Rychlost seřazení posloupnosti 100 prvků datového typu string. 69
Graf 4: Rychlost řazení posloupnosti 5.000 prvků datového typu integer. 70
Graf 5: Rychlost seřazení posloupnosti 5.000 prvků datového typu double. 70
Graf 6: Rychlost seřazení posloupnosti 5.000 prvků datového typu string. 71
Graf 7: Rychlost řazení posloupnosti 100.000 prvků datového typu integer. 72
Graf 8: Rychlost seřazení posloupnosti 100.000 prvků datového typu double. 72
Graf 9: Rychlost seřazení posloupnosti 100.000 prvků datového typu string. 73

SEZNAM TABULEK

Tab. 1. Ukázky časové složitosti pro různé hodnoty vstupních dat N..... 16

SEZNAM PŘÍLOH

PI CD-ROM obsahující knihovnu řadících algoritmů *myStorage*, zdrojové soubory a spustitelný soubor testovací aplikace *mySorting*.

PII Tabulky dosažených časů testovaných řadících algoritmů.

PŘÍLOHA P II: TABULKY DOSAŽENÝCH ČASŮ TESTOVANÝCH ŘADÍCÍCH ALGORITMŮ.

Tabulka I: Dosažené časy algoritmů při řazení posloupnosti100 prvků typu integer v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	2,134	1,406	3,070	2,134	1,351
Insertion Sort	1,160	0,046	2,129	1,160	0,095
Select Sort	1,812	0,985	1,502	1,812	1,702
Binary Insert Sort	0,814	0,048	0,780	0,814	0,000
Shaker Sort	1,544	0,063	2,918	1,544	0,030
Gnome Sort	2,837	0,015	5,985	2,837	0,031
Shell Sort	0,391	0,299	0,160	0,391	0,160
Quick Sort	1,094	3,708	2,680	1,094	3,440
Heap Sort	0,499	0,598	0,630	0,499	0,000
Merge Sort	1,190	1,146	0,800	1,190	1,570
std::sort	1,659	0,776	0,930	1,659	0,300

Tabulka II: Dosažené časy algoritmů při řazení posloupnosti100 prvků typu double v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	2,191	1,558	3,115	2,191	1,153
Insertion Sort	0,979	0,015	2,098	0,979	0,048
Select Sort	1,755	1,585	1,798	1,755	1,577
Binary Insert Sort	0,916	0,063	1,609	0,916	0,126
Shaker Sort	1,424	0,031	2,979	1,424	0,047
Gnome Sort	3,113	0,032	5,968	3,113	0,032
Shell Sort	0,425	0,155	0,328	0,425	0,262
Quick Sort	1,123	4,282	2,737	1,123	3,761
Heap Sort	0,424	0,625	0,486	0,424	0,169
Merge Sort	1,870	1,766	2,967	1,870	2,935
std::sort	1,701	0,787	1,166	1,701	0,329

Tabulka III: Dosažené časy algoritmů při řazení posloupnosti100 prvků typu string v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	12,820	4,050	21,710	12,820	3,440
Insertion Sort	4,460	0,300	8,600	4,460	0,620
Select Sort	4,400	4,370	3,740	4,400	4,390
Binary Insert Sort	3,150	0,310	7,030	3,150	0,460
Shaker Sort	10,320	0,285	21,400	10,320	0,160
Gnome Sort	11,090	0,312	20,350	11,090	0,320
Shell Sort	1,906	2,030	1,090	1,906	0,780
Quick Sort	3,017	19,830	12,650	3,017	20,580
Heap Sort	3,076	3,209	2,790	3,076	0,620
Merge Sort	13,592	13,425	12,960	13,592	16,700
std::sort	2,810	1,422	2,490	2,810	0,610

Tabulka IV: Dosažené časy algoritmů při řazení posloupnosti5000 prvků typu integer v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	5594,000	3443,800	7661,333	3656,333	3427,000
Insertion Sort	2571,000	1,860	5177,000	261,000	3,100
Select Sort	3969,000	3965,700	4031,667	3949,800	3958,667
Binary Insert Sort	1773,500	8,880	3494,667	181,400	9,200
Shaker Sort	4375,000	2,300	7468,800	540,600	1,250
Gnome Sort	7510,000	1,600	14984,000	743,900	1,550
Shell Sort	40,600	21,900	26,500	31,400	21,600
Quick Sort	68,800	5850,100	3550,000	128,000	5816,000
Heap Sort	53,100	56,300	51,400	54,600	4,700
Merge Sort	274,900	225,000	250,000	270,400	301,500
std::sort	187,100	76,700	107,800	145,200	12,600

Tabulka V: Dosažené časy algoritmů při řazení posloupnosti5000 prvků typu double v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	5656,667	3489,667	7661,667	3687,000	3463,667
Insertion Sort	2609,667	5,333	5213,667	255,000	3,000
Select Sort	4062,667	4067,333	4062,333	4067,333	4067,667
Binary Insert Sort	1786,667	10,900	3531,333	182,333	10,667
Shaker Sort	4443,000	1,600	7635,667	562,333	1,550
Gnome Sort	7583,333	1,500	15124,667	755,333	1,567
Shell Sort	40,680	23,500	31,300	31,667	22,367
Quick Sort	68,600	5965,700	3609,000	132,900	5965,700
Heap Sort	56,200	56,500	57,333	59,400	5,267
Merge Sort	320,300	367,200	354,333	342,300	316,667
std::sort	192,200	78,100	109,333	142,200	12,433

Tabulka VI: Dosažené časy algoritmů při řazení posloupnosti5000 prvků typu string v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Bubble Sort	32473,667	9734,667	54432,333	11896,000	9114,000
Insertion Sort	10276,000	26,000	20713,333	1041,333	21,333
Select Sort	9187,333	9150,667	9489,667	9187,667	9083,333
Binary Insert Sort	6786,333	52,333	13479,333	713,333	46,333
Shaker Sort	28854,333	10,333333	54358,667	3291,667	4,600
Gnome Sort	28661,333	10,667	57396,000	2864,333	3,100
Shell Sort	203,000	125,000	159,700	166,667	121,900
Quick Sort	229,000	47427,000	27292,000	635,333	47464,000
Heap Sort	297,000	313,000	286,667	312,333	35,900
Merge Sort	677,000	630,000	682,333	640,667	619,667
std::sort	291,667	130,333	171,667	250,000	23,300

Tabulka VII: Dosažené časy algoritmů při řazení posloupnosti100000 prvků typu integer v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Shell Sort	1166,667	609,667	821,800	822,667	619,667
Quick Sort	1728,000	-	-	-	-
Heap Sort	1465,400	1528,200	1416,000	1531,000	130,333
Merge Sort	10146,333	9604,333	9802,333	9645,667	9599,000
std::sort	5114,333	2128,200	2755,333	3291,667	265,500

Tabulka VIII: Dosažené časy algoritmů při řazení posloupnosti100000 prvků typu double v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Shell Sort	1161,667	640,667	817,667	823,333	630,333
Quick Sort	1755,333	-	-	-	-
Heap Sort	1495,000	1562,667	1417,000	1547,000	125,000
Merge Sort	10182,333	9609,667	9822,667	9781,000	9708,667
std::sort	5192,667	2098,667	2484,333	3318,333	266,000

Tabulka IX: Dosažené časy algoritmů při řazení posloupnosti100000 prvků typu string v milisekundách

Testovací případ/Algoritmus	Random	Sorted	Sorted (reverse)	Sorted (last 256 random)	Identical elements
Shell Sort	5813,000	3463,333	4395,667	4445,500	3429,500
Quick Sort	6328,000	-	-	-	-
Heap Sort	8322,667	8552,000	7953,000	8624,500	726,000
Merge Sort	15693,000	14505,333	15343,500	14624,500	14367,500
std::sort	8401,000	3635,333	4382,500	5851,500	476,500