PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Algoritmy číslicového třídění

2011 Milan Šumpík

Anotace

Tato práce se zabývá skupinou třídících algoritmů, která se nazývá „Algoritmy číslicového třídění“. Text i program poslouží všem, kteří se chtějí podrobněji seznámit s těmito algoritmy a chtějí pochopit principy, na kterých jsou tyto algoritmy postaveny, jak z důvodu studia, tak z důvodu uvažované implementace v nějakém projektu.

Děkuji vedoucímu bakalářské práce RNDr. Arnoštu Večerkovi za cenné rady a připomínky, za vedení práce, ochotu a věnovaný čas. Poděkování patří i mé manželce Janě za podporu ve studiu.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

1.1. Jak napsat dobrý program nebo aplikaci?

Při studiu v kurzech programování jsem občas v sobě pocítil jednu otázku: „Jak napsat opravdu dobrý program nebo aplikaci?“

Je jasné, že není snadné definovat termín „dobrý program“. Obchodník softwarové firmy „dobrý program“ bude patrně definovat trochu jinak, než jeho programátor nebo konečný uživatel. Při vývoji aplikace jde často o požadavky přicházející z různých stran a požadující nezřídka zcela protichůdné věci. Pak je na manažeru celého projektu, aby našel to nejlepší místo jejich průsečíku tak, aby všem požadavkům bylo vyhověno v maximální možné míře.

Jeden požadavek zůstane ale shodný pro všechny zůčastněné na projektu vývoje software. Jde o požadavek výkonu. Těžko se najde uživatel, programátor, obchodník či manažer, který by neměl tento požadavek. Všichni přece chtějí, aby byl program rychlý. Je přirozené, že nikdo nechce čekat například na reakci uživatelského interfejsu nebo na kontrolu pravopisu v textovém editoru. Ale zabývat se výkonem je užitečné i kvůli úsporám. U aplikace zpracovávající miliony objektů lze díky dobře navrženému algoritmu docílit, aby program běžel milionkrát rychleji. Z toho pak vyplývá odpovídající úspora na hardware stroje, na kterém aplikace běží. Proto pečlivá tvorba algoritmu je mimořádně efektivní součástí řešení rozsáhlých úloh.

Základem každého výkonného programu jsou tedy jeho algoritmy. Obecně to vypadá tak, že implementuji metodu, která již dříve byla pro řešení takového problému vymyšlena. Možností práce s algoritmem je mnoho. Každá práce samozřejmě začíná jeho studiem a následnou implementací. Ale jsou i další možné práce s algoritmem: ověřování zda pracuje tak, jak má, experimentování s jeho různými variantami, testování na různých příkladech, porovnávání s jinými algoritmy, a také zkoumání výkonu. Při vývoji aplikace není ale nutné pracovat do hloubky se všemi algoritmy daného programu. Jsou algoritmy „běžné“, které jsou sice nutné pro běh programu, ale nějak výrazně nespotřebovávají výkon počítače. Těchto „běžných“ algoritmů (funkcí, metod) je v projektu většina. Nutnost hlubší práce s kódem přichází u těch algoritmů, které výrazně potřebují výkon počítače. Při práci na projektu mohu takové algoritmy označit jako *Klíčové algoritmy*, nebo *Algoritmy, jejichž volba je kritická*.

Mezi takové klíčové algoritmy patří algoritmy *třídící* (někdy nazývány *řadící*). Účelem třídících metod je změnit uspořádání položek v souboru podle nadefinovaného pravidla. Jde například o seřazení jmen dle abecedy, čísel dle velikosti apod. Třídících algoritmů existuje mnoho a jsou rozděleny do několika skupin. V této práci se zabývám studiem, implementací a porovnáním *Algoritmů číslicového třídění*.

1.2. Studium Algoritmů číslicového třídění

Ve studiu Algoritmů číslicového třídění jsem vycházel především z knihy Roberta Sedgewicka „Algoritmy v C – části 1 - 4“[1][2]. Nikde jinde jsem nenašel tak podrobný popis těchto algoritmů - ani na internetu, ani v jiné literatuře. Robert Sedgewick je díky těmto knihám světoznámý a není bez zajímavosti, že byl žákem Donalda E. Knutha, známého počítačového vědce a autora typografického systému TEX.

R. Sedgewick v kapitole 10 knihy popisuje 4 algoritmy číslicového třídění:

- Binární quicksort
- Číslicové třídění MSD
- Třícestný číslicový quicksort
- Číslicové třídění LSD

Těmito algoritmy se také ve své práci zabývám. Kromě popisu metod, diagramů a charakteristik, je v knize R. Sedgewicka uveden kód těchto algoritmů v jazyce C. Z tohoto kódu jsem vycházel při svém psaní týchž algoritmů v jazyce C#.

Moje práce se tedy sestává z implementace těchto 4 algoritmů v jazyce C# a z výsledků jejich vzájemného porovnávání mezi sebou s různými daty na vstupu. Obsahuje také porovnání všech těchto algoritmů s klasickým třídícím algoritmem *Quicksort*.

Součástí mé práce je i didaktický a testovací program *RadixSortAlgorithms*, který názorně vysvětluje způsob fungování těchto metod a s pomocí diagramů ukazuje způsob třídění. Tato aplikace obsahuje i testovací část, která umožňuje porovnávat tyto algoritmy mezi sebou (a s algoritmem Quicksort) s různými vlastnostmi dat na vstupu.

Následující řádky tedy poslouží všem, kteří se chtějí podrobněji seznámit s Algoritmy číslicového třídění a chtějí pochopit principy, na kterých jsou algoritmy postaveny, jak z důvodu studia, tak z důvodu uvažované implementace v nějakém projektu.

K tomuto účelu předkládám tento text s popisem algoritmů a kódy všech 4 třídících algoritmů v programovacím jazyce C#, výsledky porovnávacích testů algoritmů, testovací a didaktický program včetně jeho zdrojového kódu.

2. Algoritmy číslicového třídění

Tato část obsahuje popis způsobu třídění Algoritmů číslicového třídění. Pro studium není potřeba mít nějaké významné znalosti v tomto oboru, přesto je nutné vysvětlit některé základní pojmy.

2.1. Na začátek vysvětlení některých pojmů

Objekt třídění je *soubor položek*, třídící algoritmus seřadí položky v souboru dle daného pravidla. V této práci se zabývám tříděním souboru kladných celých čísel dle jejich velikosti od nejmenšího k největšímu.

Klíč je částí položek, jsou určeny k řízení třídění. Jde tedy o tu část položky, podle které třídící metoda mění uspořádání položek.

Stabilní a *nestabilní* metoda třídění – při použití třídící metody na soubor dle více klíčů, je stabilní metoda taková, která zachová relativní uspořádání duplicitních klíčů souboru. Příklad: Mám soubor jmen žáků, který obsahuje položky jméno a ročník, setříděný abecedně dle jména. Pokud tento soubor setřídím dle ročníků, pak stabilní metoda po tomto seřazení zároveň zachová abecední uspořádání jmen (tedy v rámci jednoho ročníku budou jména abecedně setříděna). Nestabilní metoda třídění uspořádá žáky dle ročníků, ale abecední uspořádání jmen se během třídění ztratí.

2.2. Úvod do algoritmů číslicového třídění

Algoritmus číslicového třídění pojednává s klíči jako s čísly reprezentovanými v číselné soustavě o základu R (radix) pro různé hodnoty R a pracuje s jednotlivými číslicemi daných čísel.

Pokud chci vyhledat jméno v telefonním seznamu, tak obyčejně potřebuji prohledat jen několik jeho prvních písmen k tomu, abych se dostal na stránku, kde se toto jméno nachází. Dekomponuji tedy jméno (klíč) na menší části, které pak použiji k samotnému vyhledávání. Nemusím tak porovnávat celé hledané jméno s celými jednotlivými jmény v telefonním seznamu a díky tomuto postupu se vyhledávání výrazně urychlí.

Stejně tak číslicový třídící algoritmus při své práci neporovnává celé klíče, ale jen jejich části. Uvedu zde (trochu zjednodušený) příklad: Pošta pro doručování dopisů a balíků používá poštovní směrovací čísla (PSČ). Podané dopisy a balíky se na závěr každého dne roztřídí do přihrádek podle první číslice zleva PSČ, která reprezentuje kraj, do deseti přihrádek. Obsah každé přihrádky pak putuje do jednotlivých krajských měst. Na krajské poště se dopisy setřídí stejným způsobem do přihrádek podle druhé číslice PSČ a dopisy putují do jednotlivých okresních měst. Zde se třídí dle třetí číslice stejným způsobem a dopisy putují dál i podle čtvrté a páté číslice, až se dostanou k adresátovi. Uvedený příklad s poštou je

třídění o základu 10 (R = 10 = počet přihrádek) a ukazuje, proč se také číslicové třídění někdy označuje termínem přihrádkové třídění.

Existují dva zásadně odlišné přístupy k číslicovému třídění. První třída metod zahrnuje algoritmy, jež prověřují číslice klíčů v uspořádání zleva do prava, a pracují tedy s nejvýznamnější číslicí jako první (příklad: U čísla 1659 je nejvýznamější číslice 1, protože označuje počet tisíců, všechny ostatní číslice jsou méně významné). Tyto metody se obecně označují jako „číslicové třídění dle nejvýznamnější číslice“, anglicky „most-significat-digit“, zkráceně *MSD*.

Druhá třída metod číslicového třídění je odlišná: Procházejí se číslice zprava do leva a pracují nejprve s nejméně významnou číslicí (příklad: U čísla 1659 je nejméně významná číslice 9). Tyto metody se obecně označují jako „číslicové třídění dle nejméně významné číslice“, anglicky „least-significat-digit“, zkráceně *LSD*.

2.3. Základní myšlenka algoritmů číslicového třídění

Základní myšlenkou číslicové třídění je, že neporovnává celé klíče, nýbrž jen jejich části. Algoritmy číslicové třídění jsou založeny na abstraktní operaci „vyjmi i-tou číslici z klíče“.

2.4. Pojem byte pro účely studia algoritmů číslicového třídění

Bude vhodné si pro tuto práci s klíčem a jeho částmi zavést abstrakci. Počítače obecně jsou stavěny pro zpracování *bitů* po skupinách zvaných *strojová slova*, jež jsou často seskupovány z menších částí zvaných *byty*, *klíče* jsou běžně organizovány jako sekvence *bytů*. Bude pro mě pohodlné zavést, že *byte* pro mé účely nemusí mít jen 8 *bitů*, jak je to obvyklé. Proto takováto abstrakce: *byte* je sekvence *bitů* pevné délky. *Řetězec* je sekvence *bytů* proměnné délky. *Slovo* je sekvence *bytů* pevné délky. V číslicovém třídění tedy může být *klíč slovo*, nebo *řetězec*. Počet různých hodnot *bytu* je základ R (anglicky *radix*). Protože, jak již bylo řečeno, Algoritmy číslicového třídění pojednávají s *klíči* jako s čísly reprezentovanými v číselné soustavě o základu R pro různé hodnoty R, musí také délka *bytu* nabývat různých hodnot.

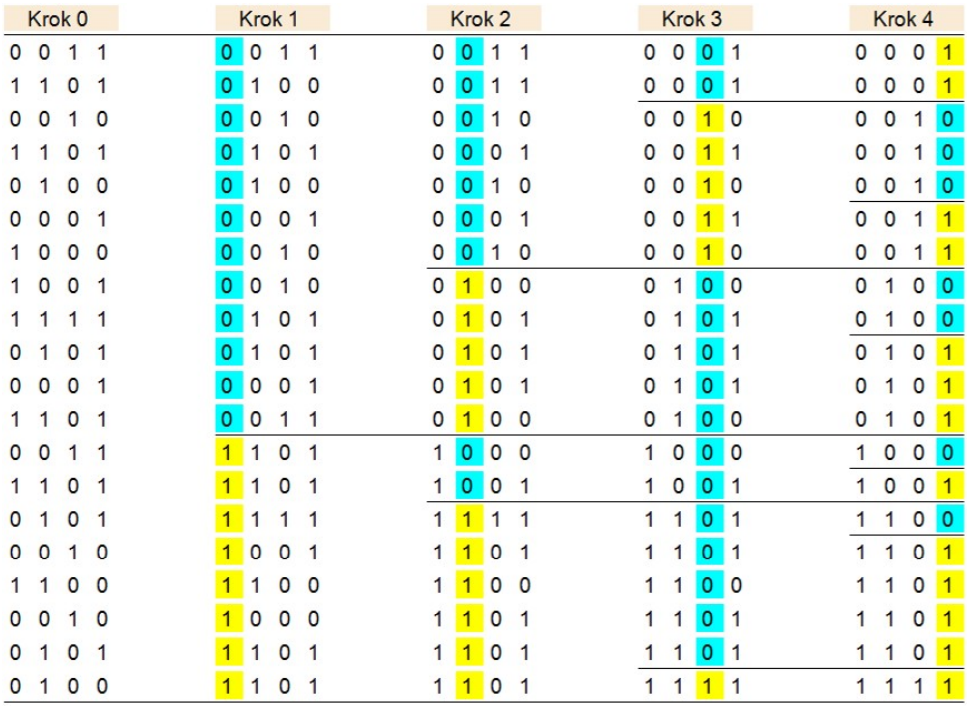
Definice klíče v číslicovém třídění: *klíč* je číslo s *radixem* (se základem) R s *číslicemi* (*byty*) očíslovány zleva, počínaje 0.

Měnit zavedený a všeobecně přijímaný pojem *byte* tak, že stanovím že jeho délka je různá, když je všeobecně vnímán vždy jako 8-bitový, se může zdát matoucí a zbytečné. Přesto přijetí toho pojmu tak, jak jsem ho zavedl (a stejně tak tento pojem zavádí R. Sedgewick ve své knize), je pro pro mě pohodlné a hlavně je důležité pro pochopení kódů algoritmů číslicového třídění.

Mohu si všimnout, že didaktická část aplikace *RadixSortAlgorithms* používá pro vysvětlení principů třídění algoritmů graf, který zobrazuje třídění 16-bitových

čísel a *byte* je v tomto případě nastaven na délku 4 *bitů* (R = 16). Toto nastavení je zvoleno tak, aby byl graf co nejpřehlednější a nejlépe vypovídal o principech algoritmů třídění. Oproti tomu testovací část aplikace třídí velká pole 32-bitových nebo 64-bitových čísel a *byte* je v tomto případě nastaven na obvyklých 8 *bitů*. Přitom jsou pro didaktickou i testovací část použity stejné algoritmy! To dokazuje, že tyto algoritmy si poradí s jakoukoliv možnou velikostí *bytu*.

Lze si také domyslet, že nastavením velikosti *bytu* lze upravovat výkon algoritmů pro konkrétní typ dat.



Obrázek 1. Diagram třídění 4-bitových čísel Binárním quicksortem (R = 2).

Vyjímkou je algoritmus *Binární quicksort*, který má délku *bytu* nastavenu vždy na hodnotu 1 (obsahuje jen 1 *bit*, R = 2), vyplývá to z jeho principu třídění. I když by se mi mohlo zdát, že by bylo jednodušší v tomto případě jednoduše hovořit o *bitu* místo o jednobitovém *bytu*, pro pochopení kódů algoritmů je lépe, když i v případě Binárního quicksortu si zvyknu hovořit o *bytu*.

Při analýze kódu algoritmů si lze všimnout, že algoritmy vždy pracují s *bytem*, a v průběhu třídění operace srovnání, na základě které dochází k přesunu prvků v poli, je vždy jen na úrovni *bytu*. Program porovnává *byte* jedné položky s *bytem* jiné položky.

Následující text odhaluje principy jednotlivých algoritmů a popisuje principy algoritmů číslicového třídění.

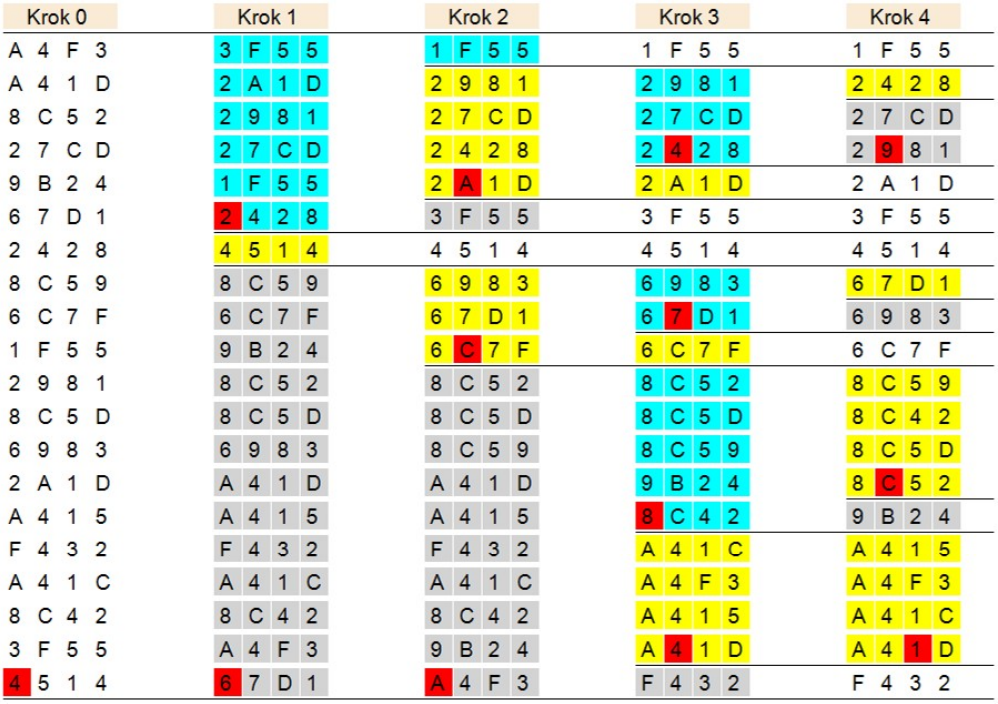
Krok 0	Krok 1	Krok 2	Krok 3	Krok 4
A 4 F 3	1 F 5 5	1 F 5 5	1 F 5 5	1 F 5 5
A 4 1 D	2 7 C D	2 4 2 8	2 4 2 8	2 4 2 8
8 C 5 2	2 4 2 8	2 7 C D	2 7 C D	2 7 C D
2 7 C D	2 9 8 1	2 9 8 1	2 9 8 1	2 9 8 1
9 B 2 4	2 A 1 D	2 A 1 D	2 A 1 D	2 A 1 D
6 7 D 1	3 F 5 5	3 F 5 5	3 F 5 5	3 F 5 5
2 4 2 8	4 5 1 4	4 5 1 4	4 5 1 4	4 5 1 4
8 C 5 9	6 7 D 1	6 7 D 1	6 7 D 1	6 7 D 1
6 C 7 F	6 C 7 F	6 9 8 3	6 9 8 3	6 9 8 3
1 F 5 5	6 9 8 3	6 C 7 F	6 C 7 F	6 C 7 F
2 9 8 1	8 C 5 2	8 C 5 2	8 C 4 2	8 C 4 2
8 C 5 D	8 C 5 9	8 C 5 9	8 C 5 2	8 C 5 2
6 9 8 3	8 C 5 D	8 C 5 D	8 C 5 9	8 C 5 9
2 A 1 D	8 C 4 2	8 C 4 2	8 C 5 D	8 C 5 D
A 4 1 5	9 B 2 4	9 B 2 4	9 B 2 4	9 B 2 4
F 4 3 2	A 4 F 3	A 4 F 3	A 4 1 D	A 4 1 5
A 4 1 C	A 4 1 D	A 4 1 D	A 4 1 5	A 4 1 C
8 C 4 2	A 4 1 5	A 4 1 5	A 4 1 C	A 4 1 D
3 F 5 5	A 4 1 C	A 4 1 C	A 4 F 3	A 4 F 3
4 5 1 4	F 4 3 2	F 4 3 2	F 4 3 2	F 4 3 2

Obrázek 2. Diagram třídění 16-bitových čísel algoritmem MSD, byte je délky 4 (R = 16) a je zapsán v šestnáckové soustavě.

2.5. Binární quicksort

Algoritmus *Binární quicksort* je variantou klasického *Quicksortu*, odtud tedy jeho název. *Byte* klíče obsahuje pouze 1 *bit*, proto je základ *bytu* roven 2 (R = 2).

Metoda třídí položky v souboru tak, že je přeskupí do pořadí, v němž všechny položky s klíči začínající nulovým *bytem* jsou před těmi, jejichž klíče začínají jedničkovým *bytem*. Soubor je tak roztříděn do (maximálně) dvou přihrádek. Třídění pokračuje dále v každé přihrádce zvlášť. V každé této přihrádce se pak přeskupí položky stejným způsobem (rekurzivně) dle druhého *bytu*. Tedy všechny záznamy v přihrádce, které mají klíč s druhým *bytem* o hodnotě 0, přesune před záznamy s druhým *bytem* o hodnotě 1. Po seřazení se pokračuje dál rekurzí (dle třetího, čtvrtého,... *bytu*), až do dosažení *bytu* posledního nebo pokud v přihrádce zůstala jen jedna položka. Tehdy je přihrádka setříděna. Soubor je setříděn po setřídění všech přihrádek.



Obrázek 3. Diagram třídění 16-bitových čísel Třícestným číslicovým quicksortem, byte je délky 4 (R = 16) a je zapsán v šestnáckové soustavě.

2.6. Číslicové třídění MSD

Předchozí metoda *Binární quicksort* třídí použitím jednoho *bitu* z klíče do dvou přihrádek, přičemž se zaměřuje nejprve na nejvýznamnější číslici. Obecně lze předpokládat, že chci třídit čísla se *základem* (radixem) *bytu* o hodnotě R, pak tedy budu potřebovat celkem R přihrádek, do kterých roztřídím soubor dle hodnoty *bytu*. Tedy například pokud bude *byte* obsahovat 8 *bitů*, pak budu k třídění potřebovat celkem 256 přihrádek (R = 256).

Princip tohoto algoritmu je tedy takový: Prochází klíči počínaje prvním, rozděluje je do přihrádek dle hodnoty *bytu* klíče, a pak obsah přihrádek rekurzivně třídí dle dalšího *bytu* v pořadí. Metoda tedy opět třídí nejprve od nejvýznamnější číslice, jak již prozrazuje její název.

2.7. Třícestný číslicový quicksort

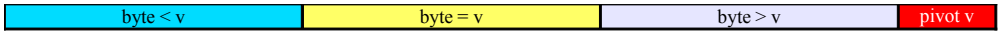
Další cestou pro přizpůsobení klasického *Quicksortu* pro číslicové třídění MSD je použití třícestného dělení. Metoda přeskupí dle vedoucího *bytu* klíčů soubor do tří podsouborů (přihrádek) dle výsledku porovnání hodnoty s předem vybraným

	Krok 5	Krok 6	Krok 7	Krok 8
	1 F 5 5	1 F 5 5	1 F 5 5	1 F 5 5
	2 4 2 8	2 4 2 8	2 4 2 8	2 4 2 8
	2 7 C D	2 7 C D	2 7 C D	2 7 C D
	2 9 8 1	2 9 8 1	2 9 8 1	2 9 8 1
	2 A 1 D	2 A 1 D	2 A 1 D	2 A 1 D
	3 F 5 5	3 F 5 5	3 F 5 5	3 F 5 5
	4 5 1 4	4 5 1 4	4 5 1 4	4 5 1 4
	6 7 D 1	6 7 D 1	6 7 D 1	6 7 D 1
	6 9 8 3	6 9 8 3	6 9 8 3	6 9 8 3
	6 C 7 F	6 C 7 F	6 C 7 F	6 C 7 F
	8 C 5 D	8 C 4 2	8 C 4 2	8 C 4 2
	8 C 4 2	8 C 5 9	8 C 5 2	8 C 5 2
	8 C 5 9	8 C 5 D	8 C 5 D	8 C 5 9
	8 C 5 2	8 C 5 2	8 C 5 9	8 C 5 D
	9 B 2 4	9 B 2 4	9 B 2 4	9 B 2 4
	A 4 1 C	A 4 1 5	A 4 1 5	A 4 1 5
	A 4 1 D	A 4 1 D	A 4 1 C	A 4 1 C
	A 4 1 5	A 4 1 C	A 4 1 D	A 4 1 D
	A 4 F 3	A 4 F 3	A 4 F 3	A 4 F 3
	F 4 3 2	F 4 3 2	F 4 3 2	F 4 3 2

Obrázek 4. Dokončení diagramu třídění Třícestným číslicovým quicksortem.

bytem kterému se obvykle říká *pivot*.

První přihrádka bude obsahovat položky s klíči, v kterých je *byte* menší než *pivot*, v prostředním podsouboru budou položky, pro jejichž *byte* v klíči platí *byte* = *pivot* a v posledním podsouboru budou ty položky souboru, jejichž *byte* v klíči je větší než *pivot*. *Pivot* samotný se po přemístění všech položek do přihrádek přiřadí k prostřednímu podsouboru.



Obrázek 5. Tři přihrádky, do kterých třídí Třícestný číslicový quicksort.

Metoda se pak rekurzivně aplikuje na podsoubory s tím, že k přesunu k následujícímu *bytu* v pořadí jako k aktuálnímu dojde pouze pro střední podsoubor.

2.8. Číslicové třídění LSD

Alternativní metodou číslicového třídění je prověřování *bytů* zprava do leva, třídí se tedy nejprve od nejméně významné číslice. Metoda začíná přeskupovat položky v souboru nejprve od posledního *bytu* a cyklus pokračuje *bytem* předchozím, dokud nedojde k *bytu* prvnímu (*bytu* číslo 0). Prochází klíči, rozděluje je do

Krok 0	Krok 1	Krok 2	Krok 3	Krok 4
A 4 F 3	6 7 D 1	4 5 1 4	A 4 1 5	1 F 5 5
A 4 1 D	2 9 8 1	A 4 1 5	A 4 1 C	2 4 2 8
8 C 5 2	8 C 5 2	A 4 1 C	A 4 1 D	2 7 C D
2 7 C D	F 4 3 2	A 4 1 D	2 4 2 8	2 9 8 1
9 B 2 4	8 C 4 2	2 A 1 D	F 4 3 2	2 A 1 D
6 7 D 1	A 4 F 3	9 B 2 4	A 4 F 3	3 F 5 5
2 4 2 8	6 9 8 3	2 4 2 8	4 5 1 4	4 5 1 4
8 C 5 9	9 B 2 4	F 4 3 2	2 7 C D	6 7 D 1
6 C 7 F	4 5 1 4	8 C 4 2	6 7 D 1	6 9 8 3
1 F 5 5	1 F 5 5	8 C 5 2	2 9 8 1	6 C 7 F
2 9 8 1	A 4 1 5	1 F 5 5	6 9 8 3	8 C 4 2
8 C 5 D	3 F 5 5	3 F 5 5	2 A 1 D	8 C 5 2
6 9 8 3	2 4 2 8	8 C 5 9	9 B 2 4	8 C 5 9
2 A 1 D	8 C 5 9	8 C 5 D	8 C 4 2	8 C 5 D
A 4 1 5	A 4 1 C	6 C 7 F	8 C 5 2	9 B 2 4
F 4 3 2	A 4 1 D	2 9 8 1	8 C 5 9	A 4 1 5
A 4 1 C	2 7 C D	6 9 8 3	8 C 5 D	A 4 1 C
8 C 4 2	8 C 5 D	2 7 C D	6 C 7 F	A 4 1 D
3 F 5 5	2 A 1 D	6 7 D 1	1 F 5 5	A 4 F 3
4 5 1 4	6 C 7 F	A 4 F 3	3 F 5 5	F 4 3 2

Obrázek 6. Diagram třídění 16-bitových čísel algoritmem LSD, byte je délky 4 (R = 16) a je zapsán v šestnáckové soustavě.

přihrádek dle hodnoty *bytu* klíče. Na rozdíl od předchozích algoritmů pokračuje v třídění dle dalšího *bytu* v celém souboru, ne jen v přihrádkách. Pochopení principu tohoto algoritmu může trvat déle, než u předchozích, není tak intuitivní. Jádrem k němu je skutečnost, že třídění musí být *stabilní*. Pokud třídění tuto vlastnost nemá, tak algoritmus LSD vůbec nefunguje.

Důkaz fungování LSD: Po uspořádání klíčů dle jejich i koncových bytů (stabilním způsobem) víme, že se každé dva klíče objeví v souboru ve správném pořadí (na základě dosud prověřených *bytů*), protože buď jsou první z jejich i koncových *bytů* různé, a v tomto případě třídění tento *byte* posune do správného pořadí, nebo jsou první z jejich koncových *bytů* stejné, a v tomto případě jsou ve správném pořadí v důsledku stability.

3. Program RadixSortAlgorithms

Program *RadixSortAlgorithms* slouží ke studiu *Algoritmů číslicového třídění*, k pochopení principů jejich řazení. Program také umí algoritmy testovat a porovnávat jejich účinnost mezi sebou a navíc s algoritmem *Quicksort*, který nepatří mezi číslicové algoritmy, ale obecně je velmi známý.

Program je rozdělen na dvě základní části: *Část didaktickou*, která slouží ke studiu způsobu třídění jednotlivých algoritmů číslicového třídění a *část testovací*, která umožňuje porovnávat výkon algoritmů mezi sebou.

3.1. Instalace aplikace a systémové požadavky

Systémové požadavky pro běh programu

Software:

- operační systém MS Windows XP SP3, MS Windows 7 (32 i 64 bitová verze)
- Internet Explorer 5.01 nebo novější
- .Net Framework 4 Client Profile

Hardware:

- 32bitový (x86) nebo 64bitový (x64) procesor s frekvencí 1 gigahertz (GHz) nebo vyšší
- 2 gigabajty (GB) paměti RAM (32bitový systém) nebo 4 GB paměti RAM (64bitový systém)

Poznámka: Pokud není v počítači nainstalován *.Net Framework 4 Client Profile*, lze ho přidat do systému hned několika způsoby. Asi nejjednoduší způsob je použít službu operačního systému *Windows update*. Nebo je možné ze stránek firmy Microsoft (www.microsoft.com/downloads) volně stáhnout instalační program a poté ho nainstalovat. *.Net Framework* je také přiložen na instalačním DVD programu.

Instalace je ve své podstatě jen rozbalení zip-archívu a začne spuštěním programu *RadixSortAlgorithmsInstall.exe*. Instalační program se zeptá na umístění, kam má program instalovat. Vyberu složku, kam chci program umístit a potvrdím. V této vybrané složce vznikne po instalačním procesu složka nová s názvem *RadixSortAlgorithms*, ve které program bude. U novějších verzí Windows (v závislosti na nastavení bezpečnosti) se může na konci instalace objevit hlášení, že program není pravděpodobně správně nainstalován. V tomto případě stačí potvrdit instalaci tlačítkem *Tento program je nainstalován správně*.

3.2. Spuštění programu

Program se spouští souborem *RadixSortAlgorithms.exe* ve složce *RadixSortAlgorithms*. Lze jej také spustit přímo z DVD bez instalace, a to ze složky *bin/RadixSortAlgorithms*.

3.3. Didaktická část programu – popis ovládání

Po spuštění programu se jako první objeví její didaktická část. Ta slouží k pochopení principů způsobu třídění algoritmů pomocí grafického znázornění přesunů položek v souboru. Tento graf nebo spíše diagram ukazuje třídění souboru o 20 položkách. Program tato vstupní data setřídí a zapamatuje si polohu položek souboru v jednotlivých krocích třídění. Ty pak zobrazí v přehledném barevném grafu a označí přihrádky. Uživatel může pak na tomto grafu studovat, jakým způsobem algoritmus pracuje, a zároveň má možnost pro lepší pochopení měnit položky vstupního pole a sledovat, jakým způsobem se pak změní i třídění.

Diagram zabírá největší část okna, který po spuštění programu zobrazuje krok číslo 0 třídění – tedy nesetříděné vstupní pole. Pole je v grafu zobrazeno jako sloupec, kde každý řádek tohoto sloupce obsahuje jeden klíč. Klíč je rozdělený na 4 *byty* – čtverce s hodnotou uvnitř. *Byte* pro naše účely nemusí mít jen 8 bitů, jak je to obvyklé (podrobněji viz. kapitola 2.4. na straně 11). Pro účely grafu je *byte* podle typu vybraného algoritmu buď 4-bitový, pak nabývá hodnot 0 – F, nebo je *byte* 1-bitový, pak nabývá hodnot 0 nebo 1. *Byty* jsou číslovány od 0 do 3, počínaje prvním zleva. Hodnoty *klíčů* u 4-bitového *bytu* nabývají hodnoty 0 - 65 535 desítkově a u 1-bitového *bytu* 0 – 15 desítkově. Jednobitový byte používá jen algoritmus *Binární quicksort*.

Graf tedy zobrazuje vstupní pole, které má 20 položek. Položka je 16-bitová, jen *Binární quicksort* pracuje s položkami, které jsou 4-bitové. Po zmáčknutí tlačítka *Start* program v grafu začne zobrazovat v určitém časovém intervalu další kroky třídění – rozmístění položek v poli v jednotlivých krocích.

Přihrádky v grafu jsou označeny vodorovnými čarami mezi klíči. Barevné značení slouží k lepší přehlednosti a snadnější identifikaci přihrádek a k identifikaci *aktuálního bytu* – tedy *bytu*, podle kterého se v daném kroku třídí. Pouze u algoritmu *Třícestný číslicový quicksort* aktuální *byte* určuje *pivot*, který je označen červeně. V tomto případě číslo aktuálního *bytu* je rovno číslu *pivota* v dané přihrádce.

Ovládání grafu je jednoduché, tlačítkem *Start* program začne v grafu zobrazovat v určitém časovém intervalu další kroky třídění – rozmístění položek v poli po jednotlivých krocích. Zobrazování kroků třídění lze kdykoliv zastavit tlačítkem *Pozastavit*. Znovu obnovit zobrazování lze tímtéž tlačítkem.

Při každém zobrazení nového kroku se v části *Popis třídění algoritmů* pro něj objeví popis toho, co se v tomto kroku děje. Pokud je třídění u konce, lze mezi těmito popisy listovat pomocí tlačítek umístěných pod textem.

Průběh třídění mohu opakovat opětovným stiskem tlačítka *Start*.

Výběr algoritmu pro zobrazení lze v pravém horním rohu pomocí přepínače. Stejná funkce je i v menu pod položkou *Třídění*.

Změna vstupních dat – program umožňuje změnit hodnoty položek vstupního pole. Dialog pro změnu pole se objeví po zmáčknutí tlačítka *Změnit vstupní data*. Hodnoty mohu zadávat jak v hexadecimální, tak v desítkové soustavě. K dispozici je i tlačítko pro rychlé naplnění pole náhodnými hodnotami.

Diagram algoritmu *Binární quicksort* z tohoto dialogu použije pro sebe jen poslední *byte*.

V tomto dialogu mohu navíc změnit i časový interval mezi zobrazením kroků v grafu. Změny se uloží a projeví jen po stisku tlačítka OK.

3.4. Program na testování výkonu algoritmů

Do testovacího režimu programu se dostanu z jeho didaktické části skrze menu: *Třídění – Testy algoritmů*. V okně určeném pro testy je třeba nejdříve zaškrtnout ty algoritmy, které chci testovat. V další části nastavím velikost pole, zda položky pole budou obsahovat 32-bitová nebo 64-bitová čísla. Dále nastavím, zda bude pole obsahovat opakující se hodnoty a procento počtu opakujících se položek. Pak už jen stačí stisknout tlačítko *Start třídění*, a tím začnou testy.

Testy probíhají tak, že se nejprve vygeneruje pole náhodných čísel s odpovídající velikostí a odpovídajícím opakováním hodnot, kterou jsem zadal. V další fázi se toto pole třídí vybranými algoritmy, jeden po druhém.

Výsledky i průběh testů se průběžně vypisují do části *Průběh třídění*. Stojí za zmínku, že se vypisují výsledky funkcí, které kontrolují stav pole. Tedy pokud se vypíše *Pole je setříděno.*, znamená to, že funkce, která kontroluje, zda je pole setříděno, vrátila kladnou odpověď. Stejným způsobem je to i s výpisem *Opakování hodnot u 0% položek.* - funkce, která kontroluje a měří opakování hodnot v poli vrátila hodnotu 0%.

Testy velmi zatěžují procesor a spotřebovávají operační paměť počítače. Proto se může stát, že okno aplikace u větší délky tříděného pole reaguje pomaleji.

Po skončení testů se výsledky zapíší do části okna s názvem *Výsledky měření*. Program obsahuje 20 pozic, do kterých výsledky měření ukládá (pokud provedu 21 test, bude zapsán do pozice 1, atd.). Přehled všech výsledků testů lze zobrazit stisknutím tlačítka *Přehled*. Spolu s výsledky je součástí přehledu i graf pro porovnání algoritmů mezi sebou.

Pro ještě lepší možnosti porovnání program umožňuje do testů zahrnout také klasický třídící algoritmus *Quicksort*, který nepatří mezi algoritmy číslicového třídění. *Quicksort* je algoritmus velmi známý a mezi programátory oblíbený, proto může být užitečné porovnat číslicové algoritmy právě s tímto třídícím programem.

Testovací formulář také umožňuje kopírovat výsledky testů do schránky operačního systému Windows. Stačí po provedených testech umístit kurzor do části okna *Průběh třídění* a pak standardními klávesovými zkratkami Windows *Ctrl+a* a potom *Ctrl+c* výsledky zkopírovat.

Práci v testovací části aplikace ukončím tlačítkem *Konec testů*. Tím program zároveň zapomene všechny naměřené výsledky testů.

4. Výsledky porovnání výkonu algoritmů číslicového třídění

V této části jsou popsány výsledky porovnání účinnosti jednotlivých algoritmů číslicového třídění mezi sebou a s algoritmem postaveným na operaci srovnání – *Quicksortem*.

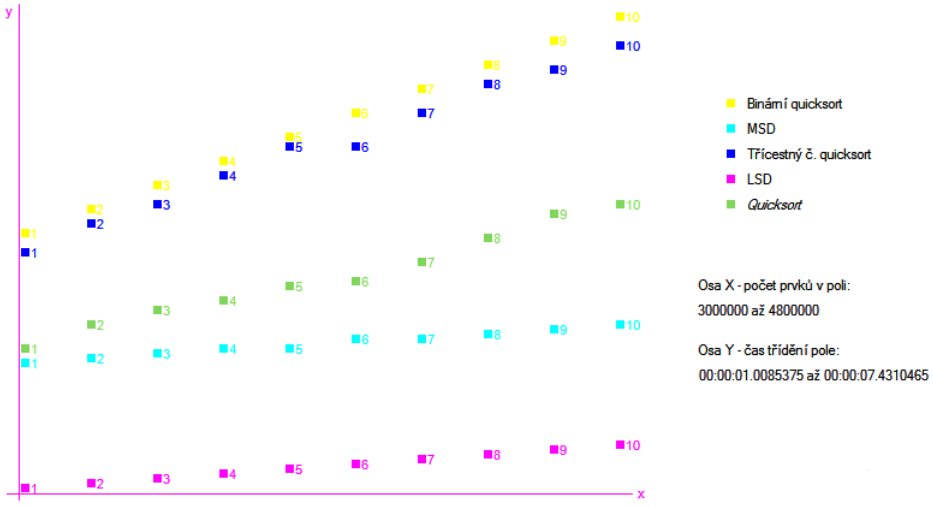
Testy byly provedeny pomocí programu *RadixSortAlgorithms* a jejich princip je takový: Program vygeneruje pole náhodných čísel, na toto pole jsou pak aplikovány postupně za sebou všechny algoritmy číslicového třídění: Binární quicksort, algoritmus MSD, Třícestný číslicový quicksort a algoritmus LSD. Délka *bytu* je nastavena, s vyjímkou Binárního quicksortu, na 8 *bitů* (R = 256). Nakonec je k setřídění použit obyčejný Quicksort. Všechny algoritmy v rámci jednoho testu třídí stejné pole. Testy jsou číslovány od 1 do 10.

Testy byly ve všech případech 10x opakovány s tím, že se postupně zvyšuje velikost tříděného pole o 200 000 položek. První test třídí pole o počtu 3 000 000 položek a poslední desátý test třídí pole o velikosti 4 800 000.

Na základě dosažených výsledků testů pak program vykreslil graf. Graf je vytvořen tak, že na osu x se nanáší hodnota velikosti tříděného pole a na osu y čas potřebný k setřídění tohoto pole.

Testování probíhalo na stroji s procesorem Intel Core i5 2,80GHz, s 8 GB DDR3 pamětí RAM a s operačním systémem MS Windows 7 Professional SP1 64-bit CZ.

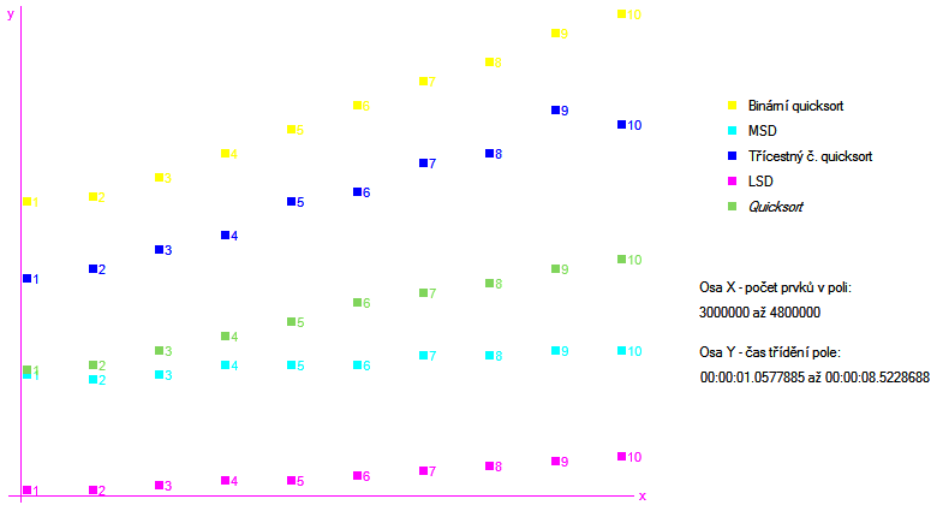
4.1. Třídění 32-bitových kladných čísel



Obrázek 7. Graf prorovnání účinnosti algoritmů při třídění 32-bitových čísel.

Z grafu je patrné, že algoritmy *Binární quicksort* a *Třícestný číslicový quicksort* v porovnání s ostatními algoritmy mají výrazně nižší výkon. Tyto algoritmy poráží i obyčejný *Quicksort*. Ten ale naopak s přibývající velikostí pole stále více ztrácí na algoritmy MSD a LSD. Vítězem tohoto klání bych označil právě tyto algoritmy: LSD i MSD. A proč oba? LSD sice obsazuje první místo, rozdíl mezi LSD a MSD se ale se stoupající velikostí pole zmenšuje.

4.2. Třídění 32-bitových kladných čísel s 40% opakování hodnot položek v poli



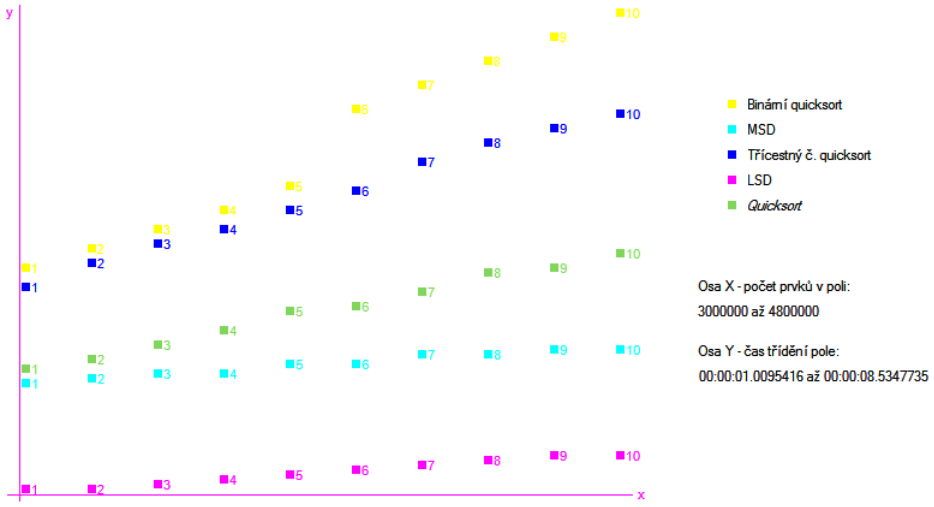
Obrázek 8. Graf prorovnání účinnosti algoritmů při třídění 32-bitových čísel s opakováním hodnot u 40% klíčů.

V tomto případě tříděné pole 32-bitových kladných čísel obsahuje 40% položek, jejichž hodnota klíče se alespoň jednou v poli opakuje. Pořadí vítězů i poražených zůstává však nezměněno. *Binární quicksort* i *Třícestný číslicový quicksort* daleko ztrácejí na ostatní algoritmy. Obyčejný *Quicksort* je poráží. V čele jsou opět algoritmy LSD a MSD.

Lze si povšimnout, že opakování hodnot nejvíce vadí *Binárnímu quicksortu*.

4.3. Třídění 32-bitových kladných čísel – důsledek změny počtu opakujících se hodnot na výkonnost algoritmů

Tento test ukazuje opět porovnání účinnosti algoritmů při třídění 32-bitových čísel, polovina testů (testy č. 1 až 5) je však provedena s polem, které neobsahuje žádné položky, které by se v poli opakovaly. Druhá polovina testů (testy č. 6 až 10) je naopak s polem, které obsahuje 40% opakujících se položek.



Obrázek 9. Graf porovnání účinnosti algoritmů při třídění 32-bitových čísel ukazuje důsledek změny počtu opakujících se hodnot z 0 na 40%.

Na uvedeném grafu jde názorně vidět, že většina algoritmů změnu v množství opakujících se položek v souboru ani nezaznamenala. Vyjímkou je *Binární quicksort*, kterému opakující se položky znatelně zhoršují výkon.

4.4. Třídění 64-bitových kladných čísel

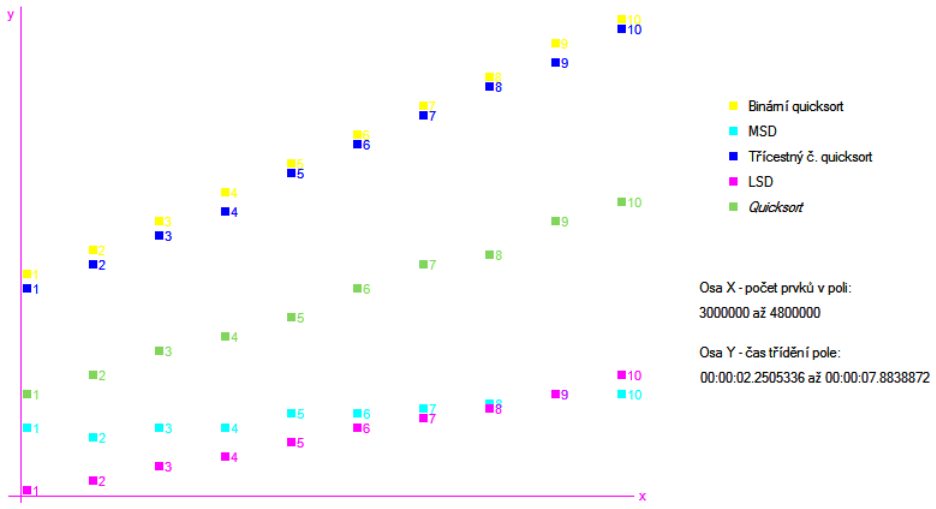
Porovnání účinnosti třídících algoritmů u pole s 64-bitovými klíči oproti 32-bitovým nepřineslo žádná překvapení. Pořadí vítězů a poražených je stále stejné. *Binární quicksort* spolu s *Třícestným číslicovým quicksortem* jsou na tom, jako obvykle, nejhůře. Setřídit 64-bitové pole jim trvá nejdéle.

Obyčejný *Quicksort* je opět uprostřed mezi poraženými a vítězi.

Nejlépe si vedli zase LSD a MSD. Je zajímavé, že LSD má s 64-bitovými klíči evidentně více práce než než 32-bitovými. V důsledku toho MSD algoritmus LSD rychleji dohání a předhnání. Přesto LSD drží s MSD krok i nadále. Je tedy spravedlivé opět prohlásit algoritmy LSD a MSD za vítěze v provnávacím testu algoritmů třídících 64-bitové pole.

4.5. Třídění 32/64-bitových kladných čísel – důsledek změny délky klíče na výkonnost algoritmů

R. Sedgewick ve své knize *Algoritmy v C* uvádí, že číslicové třídění se vyplatí tam, kde klíče jsou už dostatečně dlouhé. Pro něj jsou to klíče o délce minimálně 64 bitů. Bude tedy zajímavé porovnat účinnost algoritmů číslicového třídění na pole 32 i 64 bitové zároveň.



Obrázek 10. Graf porovnání účinnosti algoritmů při třídění 64-bitových čísel.

Následující graf na straně 25 ukazuje porovnání účinnosti algoritmů při třídění 32-bitových i 64-bitových klíčů. Polovina testů (testy č. 1 až 5) je provedena s polem které má 32-bitové klíče. Druhá polovina testů (testy č. 6 až 10) je s 64-bitovými klíči.

Graf ukazuje, že změna z 32 na 64 *bitů* se dotkla výkonu všech algoritmů. Všechny algoritmy třídí 64-bitová čísla delší dobu, než 32-bitová.

Nejvíce zvětšení delky klíčů zpomalilo algoritmus LSD. Je to přirozené, protože algoritmus LSD nepoužívá rekurzi, ale cyklus. Tento cyklus má tolik opakování, kolik je *bytů* v klíči. Velikost *bytu* je při testech nastavena na 8 *bitů*. 64-bitové klíče mají tedy 2x více *bytů* než klíče 32-bitové, tzn. 2x více cyklů.

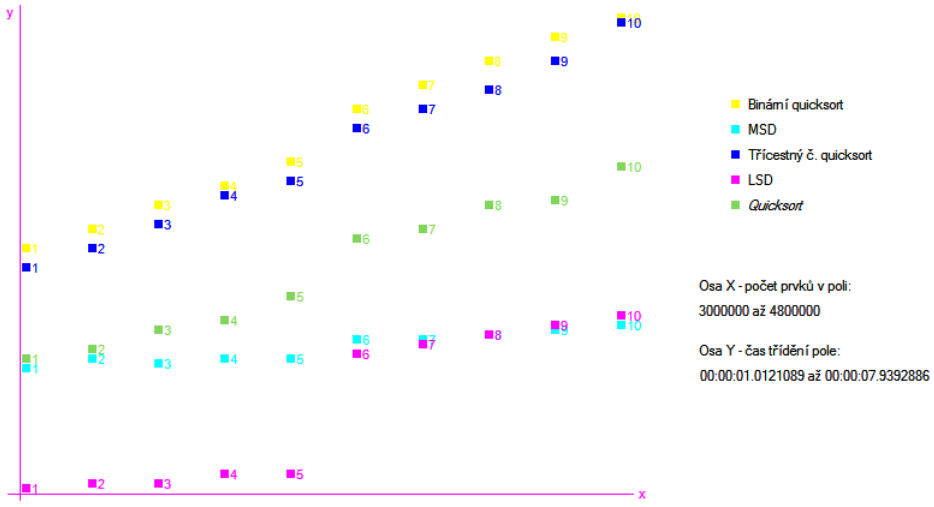
Naopak nejméně mělo prodloužení délky klíče vliv na třídění pomocí MSD, změna je zde velmi malá.

Obyčejný *Quicksort* přechod na 64-bitová čísla také výrazně zpomalil. Od vítězných algoritmů se ale také s přibývajícím počtem prvků v poli rychleji vzdaluje, než je tomu v případě 32-bitových čísel.

4.6. Třídění 64-bitových kladných čísel s 40% opakování hodnot položek v poli

V tomto případě (viz. graf na straně 26) tříděné pole 64-bitových kladných čísel obsahuje 40% položek, jejichž klíče se alespoň jednou opakují. Pořadí vítězů i poražených zůstává opět nezměněno. V čele jsou opět algoritmy LSD a MSD, nejméně výkonné jsou algoritmy *Třícestný číslicový quicksort* a *Binární quicksort* (ten je vůbec nejhorší).

Zhruba uprostřed mezi nejlepšími a nejhoršími je opět se svým výkonem oby-



Obrázek 11. Graf porovnání účinnosti algoritmů zobrazující důsledek změny délky klíče z 32 na 64 bitů.

čejný *Quicksort*.

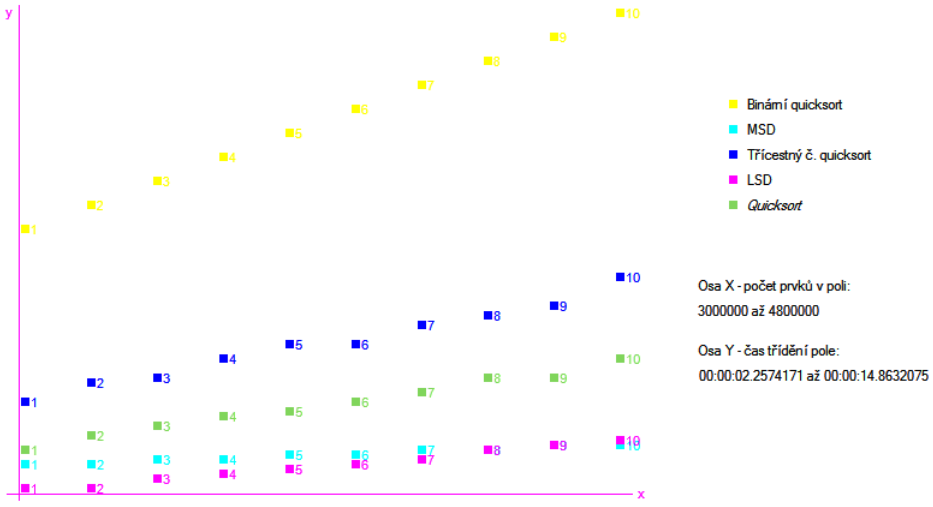
Nelze si nevšimnout že *Binární quicksort* je se svým výkonem výrazně pozadu za ostatními algoritmy. Zaostává výrazněji než v testech, ve kterých se třídilo pole bez opakujících se hodnot.

Pohled na kód algoritmu *Binární quicksort* a jeho analýza může odhalit, proč opakující se hodnoty způsobují výrazné zvýšení počtu operací srovnání. Stejné hodnoty v jedné přihrádce způsobí, že přihrádka obsahuje stále stejné množství položek i v dalších rekurzích. Limitní podmínka rekurze říká, že přihrádka je setříděna pokud v ní zůstal maximálně jeden prvek nebo pokud program prošel a roztřídil položky podle v pořadí všech *bitů* u všech položek v přihrádce. Když v přihrádce s položkami, které mají shodnou hodnotu, první část podmínky nikdy nenastane, bude muset program setřídit položky podle všech *bitů* v klíčích. A to je nesporně zdržení.

4.7. Třídění 64-bitových kladných čísel – důsledek změny počtu opakujících se hodnot na výkonnost algoritmů

Graf na str. 27 ukazuje porovnání účinnosti algoritmů při třídění 64-bitových čísel, polovina testů (testy č. 1 až 5) je provedena s polem, které neobsahuje žádné opakující se položky. Druhá polovina testů (testy č. 6 až 10) je naopak s polem, které obsahuje 40% opakujících se položek.

Z grafu jde vidět, že výkonnost algoritmů nijak zvlášť neovlivňuje opakující se hodnoty. S jedinou vyjímkou, a tou je algoritmus *Binární quicksort*. Opakování hodnot, stejně jako u 32-bitových čísel, *Binárnímu quicksortu* velmi výrazně snižuje výkon.



Obrázek 12. Graf prorovnání účinnosti algoritmů při třídění 64-bitových čísel s opakováním hodnot u 40% klíčů.

Důvod tohoto výrazného snížení je popsán v předcházející kapitole a algoritmus odsuzuje k poslednímu místu v pořadí algoritmů dle výkonu.

4.8. Shrnutí porovnání účinnosti algoritmů číslicového třídění

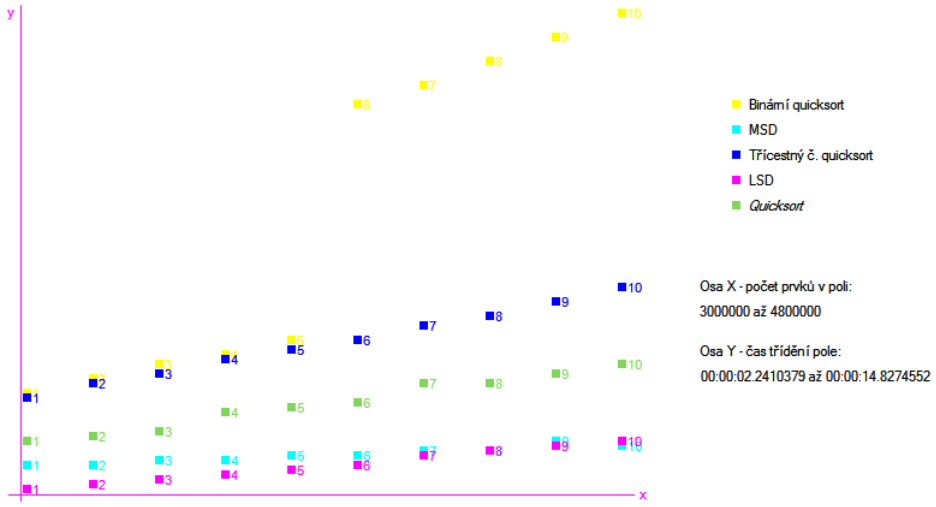
Algoritmy *Binární quicksort* a *Třícestný číslicový quicksort* mají ve srovnání s ostatními nejhorší účinnost, *Binární quicksort* navíc znatelně ztrácí na výkonu při třídění s výskytem opakujících se hodnot v tříděném souboru. Naopak nejlepší výsledky dosahují algoritmy *LSD* a *MSD*. Obyčejný *Quicksort* je se svým výkonem zhruba uprostřed mezi nejlepšími a nejhoršími.

4.9. Algoritmy číslicového třídění jsou vhodné pro pole dlouhých klíčů

Testy ukázaly, že zejména algoritmu *MSD* příliš nevadí, oproti např. *Quicksortu*, prodloužení délky klíče na dvojnásobek. Při změně délky klíče z 32-bitů na 64-bitů je zhoršení výkonu *MSD* v grafu téměř neznatelné. Naproti tomu stejná změna u všech ostatních algoritmů přinesla evidentní zhoršení výkonu.

4.10. Který algoritmus implementovat?

Volba třídícího algoritmu pro implementaci závisí především na povaze vstupních dat. Jejich vlastnosti určí nejen volbu algoritmu, ale v tomto případě i veli-



Obrázek 13. Graf porovnání účinnosti algoritmů při třídění 64-bitových čísel ukazuje důsledek změny počtu opakujících se hodnot z 0 na 40%.

kost *bytu* (tedy *radix*).

Pro implementaci bych algoritmy *Binární quicksort* a *Třícestný číslicový quicksort* vůbec nedoporučoval. Než tyto dva algoritmy, to už je lepší implementovat *Quicksort*, jehož výkon je daleko lepší než těchto dvou, a zároveň má velmi jednoduchý kód a implementace je proto snadná.

Pokud budu klást silný důraz na výkon, mohu uvažovat o algoritmech *LSD* a *MSD*. Testy ukázaly, že jsou rychlejší jak *Quicksort* ve všech případech, a zejména při třídění pole s 64-bitovými klíči.

Pokud jde o volbu mezi *LSD* a *MSD*, jejich výkon je celkem vyrovnaný. Testy ukázaly, že pořadí porovnání jejich výkonu se mění v závislosti na povaze tříděných dat. Pole s 32-bitovými klíči třídil rychleji *LSD*, ale svůj náskok před *MSD* ztrácel s přibývající délkou pole. U polí s 64-bitovými klíči tento náskok ztratil *LSD* prakticky ihned a *MSD* v tomto testu zvítězil.

Shrnuto: Pokud potřebuji třídit 32-bitová čísla a nekladu vážný důraz na výkon, uvažoval bych o implementaci *Quicksortu*. Pokud by naopak výkon byl pro mne klíčový anebo bych potřeboval třídit soubor s 64-bitovými (nebo delšími) klíči, použil bych algoritmy *LSD* nebo *MSD*. O volbě mezi těmito dvěma algoritmy bych dal rozhodnout testům nad vzorky reálných dat.

4.11. Rychlost algoritmů číslicového třídění

Algoritmy *MSD* a *LSD* v testech porazily *Quicksort*. Ten bývá zařazován mezi nejrychlejší známé řadící algoritmy. Proto i *MSD* a *LSD* mohu zařadit do stejné skupiny – mezi nejrychlejší známé algoritmy. Tím je určena i jejich užitečnost a nabízí se velké možnosti jejich uplatnění v různých projektech.

5. Popis kódu v jazyce C# společného všem třídícím algoritmům

5.1. Vývojové prostředí a verze jazyka

Algoritmy číslicového třídění byly napsány v jazyce C# verze 4.0 ve vývojovém prostředí Microsoft Visual Studio 2010. Program ke svému běhu potřebuje *.Net Framework 4 Client Profile.*

5.2. Předloha kódů a názvy proměnných a metod

Při psaní jsem se držel kódů algoritmů uvedených v knihách R. Sedgewicka, které jsou zde napsány v jazyce C (případně C++). Při převodu těchto kódů do jazyka C# to pro mě znamenalo na prvním místě kódy v knize nastudovat. Uvedené programy v knihách obsahují několik drobných chyb, které neodhaleny způsobují, že algoritmy špatně fungují. Studium těchto kódů bylo pro mne programátorskou školou. Musel jsem uznat, že kódy, které bych napsal pro uvedené algoritmy já, by nebyly ani tak elegantní a prosté, a zcela jistě složitější, a tím i méně výkonné.

Při přepisu kódů do jazyka C# jsem se snažil označovat názvy proměnných i metod stejně jako v knize, aby srovnání mých kódů s kódy v knize bylo co nejjednodušší. Jen v několika případech, kdy se mi můj název zdál pro přehlednost lepší, jsem jej změnil. Například *radix* (česky *základ*) je v kódech knihy označen velkým písmenem R, v mém kódu je označen názvem `radix`. Také proměnnou, která určuje limit počtu prvků v poli, při kterém se použije jednodušší třídění a v knize je označena jako M, jsem nazval `limitForEasySort`. Ve zdrojovém kódu aplikace *RadixSortAlgorithms* jsem se snažil důležité pasáže opatřit vysvětlujícími poznámkami, které by měli pomoci k jeho pochopení.

5.3. Volání třídících metod a jejich parametry

Třídící metody mají většinou tyto parametry:

- a je označeno pole, které bude tříděno
- l je označení počátečního indexu, od kterého bude pole tříděno (včetně)
- r je označení konečného indexu, do kterého bude pole tříděno (včetně)
- w označuje číslo bytu, podle kterého bude třídění probíhat

Slovní přepis volání metody třídění by tedy zněl takto: Setřiď pole a dle bytu w v klíči, a to od (levého) indexu l do (pravého) indexu r.

Například deklarace metody MSD vypadá takto:

```
void MSD(int[] a, int l, int r, int w)
      {...
```

Volání `MSD( p, 100, 10000, 0);` mohu slovy tedy přepsat takto: Setřiď, pomocí algoritmu *MSD*, pole *p* od indexu *100* do indexu *10 000* dle prvního *bytu* zleva (neboli *bytu 0*).

K úplnému porozumění kódu je třeba si na začátek objasnit značení *bitů* a *bytů* v klíči.

5.4. Způsob značení bitů a bytů v klíči

Číslicové třídění třídí soubor položek. Každá položka obsahuje klíč, podle kterého se třídí. Pro algoritmy číslicového třídění přesněji dle *bytu* klíče. Přitom platí, jak jsem již uvedl výše, že velikost *bytu* není striktně určena (délka *bytu* nemusí být 8 *bitů*, jak je obvyklé).

Při studiu kódu je nutné, aby byl jasný způsob označení *bitů* a *bytů* v klíči. Obojí je značeno číslicemi počínaje 0 zleva, od nejvýznamnější k méně významné. Tedy *byte* č. 0 je nejvýznamější.

Algoritmy číslicového třídění třídí položky dle jednotlivých *bytů*. *Aktuální byte* je ten, podle kterého se v dané chvíli třídí.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Byte	**0**				**1**				**2**				**3**			
Slovo																

Tabulka 1. Značení bitů a bytů pro účely číslicového třídění na příkladu 16-bitového slova s 4-bitovým bytem.

5.5. Hierarchie tříd

Základní třídy pro jednotlivé algoritmy jsou pojmenovány intuitivně:

- `QuickSortB` – třída pro třídění pomocí algoritmu *Binární quicksort*
- `MSDSort` – třída pro třídění pomocí algoritmu *MSD*
- `ThreeWayRadixQuicksort` – třída pro třídění pomocí algoritmu *Třícestný číslicový quicksort*
- `LSDSort` – třída pro třídění pomocí algoritmu *Číslicové třídění LSD*

Rodičovská třída: Všechny tyto základní třídy jsou potomky třídy `RadixSort`.

5.6. Seznam důležitých proměnných a metod ve třídě RadixSort

Proměnné:

- `bitsWord` – určuje počet *bitů* v klíči
- `bitsByte` – určuje počet *bitů* v *bytu*
- `bytesWord` – počet *bytu* v klíči (platí `bytesWord = bitsWord / bitsByte`)
- `radix` – radix R neboli základ, počet všech možných hodnot *bytu*
- `limitForEasySort` – určuje počet prvků v tříděném poli, od kterého se použije jednodušší třídění

Pole: `aux` – pomocné (odkládací) pole, určené ke kopii pole tříděného. Má tedy stejnou velikost jako tříděné pole.

Některé důležité metody:

- `Digit(a, b)` – funkce vrací *byte* číslo b z klíče a
- `Less(a, b)` – vrací *true*, když b je před a
- `Exch([] p, a, b)` – vymění mezi sebou prvky a a b v poli p
- `Compexch([] pole, a, b)` – pokud `Less(a, b)` vrátí *true*, pak a a b mezi sebou vymění
- `Insertion([] p, l, r)` – setřídí pole p od indexu l do indexu r pomocí *Vkládacího třídění*.

Vkládací třídění patří mezi elementární metody třídění. V implementaci číslicového třídění je použita v případech, kdy počet prvků pole v přihrádce určené k třídění klesne pod určitou mez (hodnota `limitForEasySort`). V takovém případě je výhodnější třídit pole metodou jednoduchou, která nespotřebuje další paměť rekurzí.

Třídění pak probíhá takto: Algoritmy číslicového třídění rozdělí soubor do přihrádek a ty pak třídí; pokud se stane, že v některé přihrádce je položek méně než stanovuje proměnná `limitForEasySort`, roztřídí všechny položky v přihrádce pomocí algoritmu *Vkládacího třídění* (a tím je třídění v dané přihrádce u konce).

6. Popis kódu třídících algoritmů v jazyku C#

Metody jsou napsány tak, že jsou schopny třídit pole kladných celých čísel. Potenciál algoritmů je ale větší. Nevelkou úpravu kódu by si například vyžádala uprava algoritmů pro třídění textových položek podle abecedy. Mimo jiné by to obnášelo úpravu metody *Less*, která by pracovala se znaky na vstupu.

Popsané metody jsou součástí potomků třídy `RadixSort`:

- `QuickSortB` – pro *Binární quicksort*
- `MSDSort` – pro *MSD*
- `ThreeWayRadixQuicksort` – pro *Třícestný číslicový quicksort*
- `LSDSort` – pro *Číslicové třídění LSD*

6.1. Binární quicksort

(metoda *QuickB* ve třídě *QuickSortB*)

V tomto třídění je počet *bitů* v *bytu* pevně stanoven na 1, *byte* tedy obsahuje pouze jeden *bit*. Základ (radix) *bytu* je 2.

Aktuální *byte* je na začátku třídění nastaven na číslo 0, s každou další rekurzí se číslo zvyšuje o 1.

Program v cyklu *while* prochází pole zleva i zprava. Ukazatel procházení pole zleva je proměnná i, zprava j. Cyklus končí, pokud se ukazatele i a j potkají. Při procházení zleva se kontrolují aktuální *byty* a procházení se zastaví, pokud $byte = 1$. Procházení zprava se zastaví, pokud aktuální $byte = 0$ a prvky pole s indexy i a j se mezi sebou prohodí.

Po ukončení cyklu vzniknou v poli dvě podmnožiny prvků, neboli přihrádky. Je možné, že jedna podmnožina může obsahovat 0 prvků, pokud jsou všechny *aktuální byty* v poli shodné. Jedna přihrádka bude obsahovat prvky s klíči začínající *bytem* o hodnotě 0, v druhé přihrádce budou jen prvky pole začínající *bytem* 1. První přihrádka bude mít rozsah indexů l až $(j - 1)$ a druhá přihrádka j až r.

Program pak dále pokračuje rekurzivním voláním pro každou přihrádku zvlášť, s tříděním dle dalšího *bytu* v pořadí zleva.

Rekurze se ukončí tehdy, pokud má přihrádka pouze jeden prvek nebo pokud číslo aktuálního *bytu* pro třídění překročilo celkový počet *bytů* v klíči.

6.2. Číslicové třídění MSD

(metoda *MSD* ve třídě *MSDSort*)

Tento program potřebuje k svému třídění lokální pole *count* pro uložení pozic přihrádek, a také pole *aux* určené pro dočasnou kopii pole tříděného. Pole *aux* by mělo být globální.

Aktuální *byte* je na začátku třídění nastaven na číslo 0, s každou další rekurzí se číslo zvyšuje o 1.

Třídění probíhá jako sled několika cyklů. První cyklus slouží jen k nulování pole *count*. Další kód obsahuje postup, kterému se říká *Čítání indexových klíčů* a používá se i v jiných třídících algoritmech. Jeho první cyklus ukládá do pole *count* počty výskytů jednotlivých hodnot aktuálních *bytů* klíče tímto způsobem:

ount[0] = zůstane *nulový*
count[1] = počet *bytů* o hodnotě *0*
count[2] = počet *bytů* o hodnotě *1*
count[3] = počet *bytů* o hodnotě *2*
. . .
count[radix] = počet *bytů* o hodnotě *radix – 1*

Po tomto cyklu tedy program zná počty všech hodnot aktuálních *bytů* klíčů a má je uloženy do pole *count*. Pokud chci zjistit počet klíčů s aktuálním *bytem* o hodnotě *x*, najdu nyní odpověď v hodnotě prvku pole *count[x +1]*. Jednoduše: Program zná velikosti všech přihrádek.

Dalším úkolem je zjistit indexy, od kterých budou jednotlivé přihrádky začínat v setříděném poli. Protože zná velikost přihrádek, stačí k tomu prostý součet těchto velikostí (další cyklus *for*).

V této chvíli je první index všech přihrádek obsahující *byte* hodnoty *x* uložen v poli *count* s indexem *x* (*count[x]*). Nyní program prochází tříděné pole (další cyklus) a každý jeho prvek umístí do správné přihrádky, k tomu používá pomocné pole *aux*.

Další cyklus je jen přepis pomocného pole *aux* zpět do pole původního *a*.

Nakonec program rekurzí volá sám sebe pro každou jednotlivou přihrádku zvlášť, s tříděním dle dalšího *bytu* zleva (aktuální *byte +1*). Rekurze končí po setřídění podle posledního *bytu* klíče, nebo pokud klesne počet položek v přihrádce pod určenou mez (*limitForEasySort*) zavoláním jednoduchého vkládacího třídění.

6.3. Třícestný číslicový quicksort

(metoda *QuickSort3* ve třídě *ThreeWayRadixQuicksort*)

Aktuální *byte* je na začátku třídění nastaven na číslo *0*, s každou další rekurzí se číslo zvyšuje o *1* jen u *přihrádky 2*.

Metoda třídí do tří přihrádek na základě porovnání s *pivotem*. *Pivota* si program najde v posledním prvku tříděného pole. První přihrádka (*přihrádka 1*) obsahuje položky pole, jejichž *byte* klíče je menší než *pivot*. Prostřední přihrádka (*přihrádka 2*) obsahuje *byte* = *pivot* a pro poslední třetí přihrádku (*přihrádka 3*) platí *byte* > *pivot* (samotný *pivot* se přemístí do prostřední přihrádky).

K tomuto přeskládání prvků do tří přihrádek program dojde ve dvou fázích. V první fázi jsou prvky *přihrádky 2* přemísťovány na začátek i na konec pole - do levé a pravé *přihrádky 2*. Za levou *přihrádkou 2* se skládají prvky *přihrádky*

	byte = v	byte <v	nesetříděno	byte >v	byte = v	pivot
Přihrádka	2	1	-	3	2	2
Index	l	p	i	j	q	r

Tabulka 2. Fáze 1 třídění třícestným quicksortem - tabulka ukazuje strukturu tříděného pole. Prvky patřící do přihrádky č. 2 jsou přemísťovány jak na začátek, tak i na konec pole.

	byte<v	byte=v	byte>v
Přihrádka	1	2	3

Tabulka 3. Fáze 2 třídění třícestným quicksortem - struktura pole na konci jednoho kroku třídění. V této fázi se spojí obě části přihrádky č. 2 a přemístí se doprostřed (včetně pivotu).

1 a před pravou *přihrádkou 2* prvky *přihrádky 3* a uprostřed mezi přihrádkou *1* a *3* je ještě nesetříděné pole - viz. Tabulka 2.

Z tabulky je zřejmý i význam proměnných: i obsahuje index procházení pole zprava, j index procházení zleva. Indexy *přihrádky 2* jsou dva, protože přihrádka je rozdělená: p představuje index konec levé a q začátek pravé.

V cyklu *while* tedy prochází program tříděné pole, dokud zleva nenajde prvek, pro nějž platí $byte \geq pivot$ a zprava $byte \leq pivot$. Pak oba tyto prvky vymění mezi sebou. Pokud některý z těchto dvou prvků je roven *pivotu*, znovu ho přemístí buď do levé nebo pravé *přihrádky 2*. Cyklus končí, pokud se indexy procházení zleva i zprava (i a j) překříží.

Na závěr se ve dvou cyklech *for* spojí levá i pravá část v jednu a přemístí se doprostřed, viz. Tabulka 3.

Úplně naposledy je rekurzivní volání pro všechny tři přihrádky s tím, že jen u *přihrádky 2* se přechází na další *byte* v pořadí. Rekurze končí po setřídění podle posledního *bytu*. Stejně jako předchozí, i tato implementace obsahuje aplikaci jednoduchého třídění pro nízké počty prvků v tříděné přihrádce.

6.4. Číslicové třídění LSD

(metoda *LSD* ve třídě *LSDSort*)

Zvláštností číslicového třídění *LSD* oproti předchozím algoritmům je, že program netřídí od prvního *bytu* k poslednímu, ale od *bytu* posledního k prvnímu. Další rozdíl je v tom, že nepoužívá rekurzi.

Aktuální *byte* je na začátku třídění nastaven na číslo posledního *bytu* (*bytesWord – 1*), které hlavní cyklus *for* postupně snižuje až k hodnotě *0*, a tak

prochází postupně všemi *byty*.

Uvnitř tohoto cyklu je, podobně jako v kódu algoritmu *MSD*, *Čítání indexových klíčů*. Do pomocného pole *count* se nejprve uloží počty výskytů všech hodnot aktuálních *bytů* tříděného pole, poté se určí počáteční indexy všech přihrádek sčítáním (způsobem jedna hodnota – jedna přihrádka), a podle těchto hodnot se tříděné pole zkopíruje do pomocného pole *aux*. Poslední cyklus *for* zkopíruje již setříděné pole z pomocného pole *aux* zpět do původního pole *a*. Podrobněji jsem popsal *Čítání indexových klíčů* v kapitole popisující kód algoritmu *MSD* na straně 32. Cyklus se pak opakuje s dalším *bytem* zprava.

Program neobsahuje rekurzi, ale v jednom třídění prochází tříděné pole a setřídí ho postupně podle všech *bytů*. Kroků třídění je tedy vždy tolik, kolik je *bytů* v klíči.

7. Algoritmy číslicového třídění v kódu jazyka C#

7.1. Základní metody rodičovské třídy RadixSort

```
protected UInt64 Digit(dynamic a, int b)
        {
            return (((a) >> (bitsWord-((b) + 1)*bitsByte)) & (radix - 1));
        }

protected bool Less(dynamic a, dynamic b) { return (a < b); }

static internal void Exch<T>(T[] pole, int a, int b)
        {
            T pom = pole[a];
            pole[a] = pole[b];
            pole[b] = pom;
        }

protected void Compexch<T>(T[] pole, int a, int b)
        {
            if (Less(pole[a], pole[b])) Exch<T>(pole, a, b);
        }

protected void Insertion<T>(T[] pole, int l, int r)
        {
            for (int i = l + 1; i <= r; i++) Compexch(pole, i, l);
            for (int i = l + 2; i <= r; i++)
            {
                int j = i;
                T v = pole[i];
                while (Less(v, pole[j - 1]))
                {
                    pole[j] = pole[j - 1];
                    j--;
                }
                pole[j] = v;
            }
        }
```

7.2. Kód algoritmu Binární quicksort

```
private void QuickB<T>(T[] a, int l, int r, int w)
        {
            int i = l;
            int j = r;

            if (r <= l || w >= bitsWord) return;

            while (j != i)
            {
                while (Digit(a[i], w) == 0 && i < j) i++;
                while (Digit(a[j], w) == 1 && j > i) j--;
                Exch(a, i, j);
            }

            if (Digit(a[r], w) == 0) j++;

            QuickB<T>(a, l, j - 1, w + 1);
            QuickB<T>(a, j, r, w + 1);
        }
```

7.3. Kód algoritmu MSD

```
void MSD<T>(T[] a, int l, int r, int w, T[] aux)
        {
            int[] count = new int[radix+1];
            int i, j;

            if (w >= bytesWord) return;

            if (r - l <= limitForEasySort)
            {
                Insertion(a, l, r);
                return;
            }

            for (j = 0; j < radix; j++) count[j] = 0;

            for (i = l; i <= r; i++) count[Digit(a[i], w) + 1]++;

            for (j = 1; j < radix; j++) count[j] += count[j - 1];

            for (i = l; i <= r; i++) aux[count[Digit(a[i], w)]++] = a[i];

            for (i = l; i <= r; i++) a[i] = aux[i - l];

            MSD<T>(a, l, ((l + count[0]) - 1), w + 1, aux);

            for (j = 0; j < radix - 1; j++)
             MSD<T>(a, (l + count[j]), (l + count[j + 1]) - 1, w + 1, aux);
        }
```

7.4. Kód pro algoritmus Třícestný číslicový quicksort

```
void QuickSort3<T>(T[] a, int l, int r, int D)
        {
            if (D >= bytesWord) return;

            int i, j, k, p, q;
            UInt64 v;

            if (r - l <= limitForEasySort)
            {
                Insertion(a, l, r);
                return;
            }

            v = Digit(a[r], D);
            i = l - 1;
            j = r;
            p = l - 1;
            q = r;

            while (i < j)
            {
                while (Digit(a[++i], D) < v) ;
                while (v < Digit(a[--j], D)) if (j == l) break;

                if (i > j) break;

                Exch(a, i, j);

                if (Digit(a[i], D) == v) { p++; Exch(a, p, i); }
                if (v == Digit(a[j], D)) { q--; Exch(a, j, q); }
            }

            if (p == q)
            {
                QuickSort3(a, l, r, D + 1);
                return;
            }

            if (Digit(a[i], D) < v) i++;

            for (k = l; k <= p; k++) { Exch(a, k, j); j--; }
```

```
    for (k = r; k >= q; k--) { Exch(a, k, i); i++; }

    QuickSort3(a, l, j, D);

    if (i == r && Digit(a[i], D) == v) i++;

    QuickSort3(a, j + 1, i - 1, D + 1);
    QuickSort3(a, i, r, D);
}
```

7.5. Kód pro algoritmus LSD

```
void LSD<T>(T[] a, int l, int r, T[] aux)
        {
            int i, j, w;
            int[] count = new int[radix + 1];

            for (w = bytesWord - 1; w >= 0; w--)
            {
                for (j = 0; j < radix; j++) count[j]=0;

                for (i = l; i <= r; i++) count[Digit(a[i], w) + 1]++;

                for (j = 1; j < radix; j++) count[j] += count[j - 1];

                for (i = l; i <= r; i++)
                                    aux[count[Digit(a[i], w)]++] = a[i];

                for (i = l; i <= r; i++) a[i] = aux[i - l];
            }
        }
```

7.6. Kód pro algoritmus Quicksort

Tento algoritmus **nepatří** do rodiny algoritmů číslicového třídění. Je zde uveden proto, že je součástí testů v programu *RadixSortAlgorithms* kvůli porovnání účinnosti těchto algoritmů s nějakým obecně známým a často používaným algoritmem. Mezi takové algoritmy patří mj. právě *Quicksort*.

```
static internal void Sort<T>(T[] pole, int l, int r)
        {
            if (r <= l) return;

            int i = Partition<T>(pole, l, r);

            Sort<T>(pole, l, i - 1);
            Sort<T>(pole, i + 1, r);
        }



        static internal int Partition<T>(T[] pole, int l, int r)
        {
            int i = l - 1;
            int j = r;
            T pivot = pole[r];

            for (; ; )
            {
                while (Less(pole[++i], pivot)) ;
                while (Less(pivot, pole[--j]))
                {
                    if (j == l) break;
                }

                if (i >= j) break;

                Exch<T>(pole, i, j);
            }

            Exch<T>(pole, i, r);

            return i;
        }
```

8. Dokumentace kódu aplikace RadixSortAlgorithms

Tato kapitola obsahuje popis hlavních tříd a některých jejich důležitých metod zdrojového kódu aplikace *RadixSortAlgorithms*. Měla by pomoci každému, kdo by se chtěl orientovat ve zdrojovém kódu tohoto programu, ať už z důvodu studia, nebo z důvodu úpravy kódu pro vlastní potřeby.

8.1. Vývojové prostředí a verze jazyka

Program *RadixSortAlgorithms* byl napsán v jazyce C# verze 4.0 ve vývojovém prostředí Microsoft Visual Studio 2010. Program ke svému běhu potřebuje *.Net Framework 4 Client Profile.*

8.2. Rozdělení tříd podle účelu

Všechny třídy programu lze rozdělit dle účelu na tři hlavní skupiny:

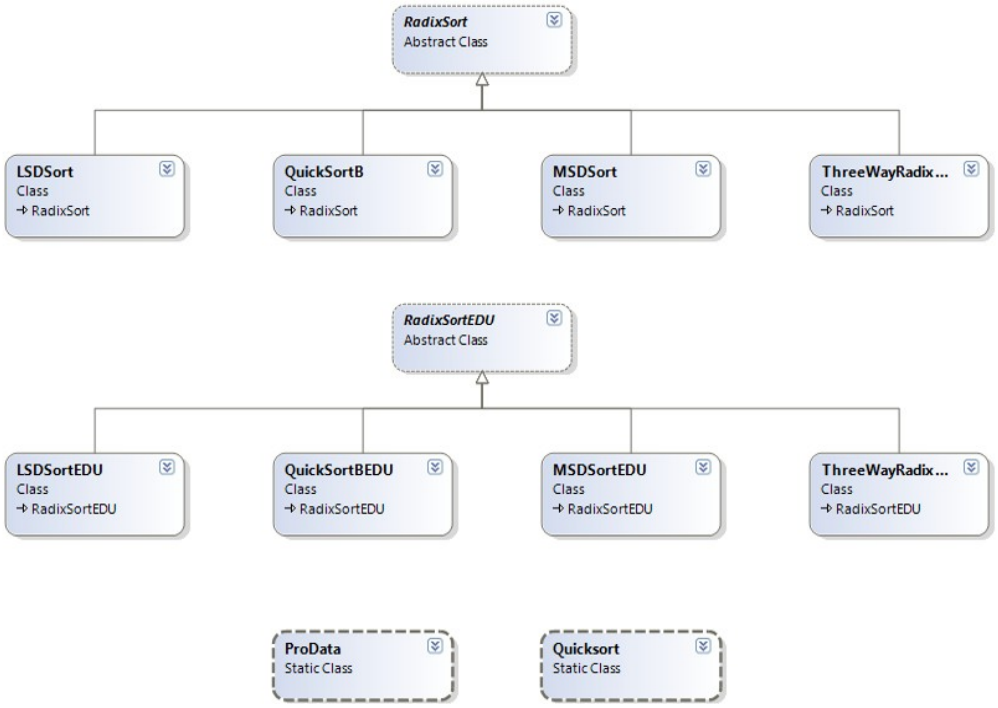
- třídy určené ke zpracování dat
- třídy uživatelského rozhraní
- třídy sloužící programu (jde o třídy *Program* a *Resources*)

8.3. Třídy pro zpracování dat

Někoho by mohlo překvapit, že číslicové třídění je zastoupeno dvěma rodičovskými třídami: *RadixSort* a *RadixSortEDU*. Tyto třídy obsahují stejné třídící algoritmy, ale liší se svými výstupy.

Třída *RadixSort* (přesněji její potomci) má na svém výstupu pouze setříděné pole, je tedy určena k třídění a právě v ní je realizována implementace algoritmů číslicového třídění. Instance potomků této třídy je určena k co nejrychlejšímu třídění a v programu se právě tyto instance používají v testech algoritmů. Tato třída je podrobněji popsána v kapitolách 5., 6. a 7.

Oproti tomu třída *RadixSortEDU* není určena k třídění, přestože obsahuje stejné algoritmy, jako třída *RadixSort*. Tyto algoritmy jsou však upraveny tak, že umožňují zobrazení didaktických diagramů k pochopení vlastních principů třídění. Jsou tedy rozšířeny o výstupy stavu tříděného souboru v jednotlivých krocích třídění a program tuto třídu (její potomky) používá ve své didaktické části. Kromě uvedených výstupů je v těchto algoritmech upravena limitní podmínka tak, aby nedocházelo k třídění pomocí *Vkládacího třídění* v případech, kdy počet prvků pole v přihrádce určené k třídění klesne pod určitou mez. Tzn., že algoritmy třídí „číslicovým způsobem“ v přihrádkách až do jejich úplného setřídění.



Obrázek 14. Třídy určené ke zpracování dat.

Třída *Quicksort* slouží k třídění souborů pomocí stejnojmenného algoritmu a program ji používá v rámci testů algoritmů.

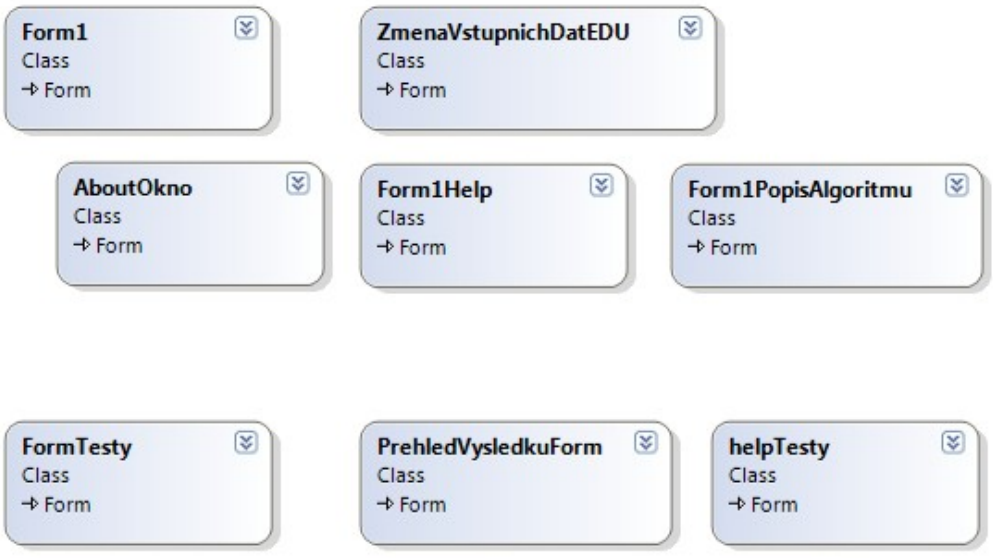
ProData je třída, která má několik metod pro zpracování tříděných dat. Právě v této třídě se plní pole náhodnými čísly (i s případným opakováním). V této třídě jsou také funkce, které zjistí, zda je pole setříděné, nebo ne a jaké obsahuje procento opakujících se hodnot. Jsou zde také metody pro převod čísel mezi hexadecimální a desítkovou soustavou a validaci hodnot pro danou soustavu.

Popis některých důležitých metod třídy ProData:

- `KopirujPole<T>(T[] a, T[] b)`- zkopíruje pole a do pole b.
- `NaplnPole(pole)` – naplní pole nahodnými čísli.
- `MyRnd16()` - generátor náhodného čísla typu UInt16.
- `MyRnd()` - generátor náhodného čísla typu UInt32.
- `MyRnd64()` - generátor náhodného čísla typu UInt64.

- `NaplnPoleRedund<T>(T[] naplnenePole, int procentoRedundance)` – do pole naplněného náhodnými čísli přidá opakující se hodnoty. Procento opakujících se hodnot na výstupu je rovno hodnotě *procentoRedundance*.
- `PoleSetrideno(UInt32[] pole)` – funkce, která vrací *true*, když je pole setříděno.
- `ProcentoRedundance<T>(T[] setridenePole)` – funkce, která spočítá procento položek v souboru, jejichž hodnota se alespoň jednou v souboru vyskytuje (opakující se data).
- `ConvertToUInt16(string retezec)` – převádí hexadecimální číslo na desítkové.
- `ValidateHex(string hexretezec)` – vrací true, pokud na vstupu je hexadecimální číslo.
- `Hexa(UInt16 cislo)` – převádí 4-bitové číslo na hexadecimální znak.

8.4. Uživatelské rozhraní



Obrázek 15. Třídy uživatelského rozhraní.

Ve skupině uživatelského rozhraní jsou dva hlavní formuláře: *Form1* a *FormTesty*. *Form1* je okno, které se objeví ihned po spuštění programu – okno, které zobrazuje diagram třídění. Jde tedy o didaktickou část programu. Z tohoto okna jde jednak vyvolat nápovědu – třídy *Form1Help*, *Form1PopisAlgoritmu*, *AboutOkno* – jednak zobrazit dialog pro změnu vstupních dat – *ZmenaVstupnichDatEDU*.

Formulář *FormTesty*, jak už název napovídá, je okno obsahující uživatelské rozhraní pro nastavení, ovládání a zobrazení výsledků testů algoritmů. Pro podrobnejší zobrazení testů může být z něj vyvolán formulář *PrehledVysledkuForm* a pro nápovědu okno *helpTesty*.

Závěr

Základní myšlenkou *Algoritmů číslicového třídění* je, že při řazení položek neporovnávají celé klíče, ale jen jejich části.

Testy ukázaly, že z této skupiny algoritmů jsou nejrychlejší algoritmy *LSD* a *MSD*, které svojí účinností předhánějí známý algoritmus *Quicksort*. Jejich náskok před *Quicksortem* se ještě zvětší, když se délka klíčů prodlouží z 32 bitů na 64. Naopak algoritmy *Binární quicksort* a *Třícestný číslicový quicksort* jsou pomalejší, než obyčejný *Quicksort*. Navíc účinnost *Binárního quicksortu* se dále výrazně zhoršuje, pokud se hodnoty klíčů v tříděném souboru opakují.

Conclusions

The basic idea of *Radix sort algorithms* is not to compare whole keys but only part of them.

The tests showed that *LSD* and *MSD* algorithms are the fastest of this group and that they are even more efficient than well-known algorithm *Quicksort*. The gap in efficiency between *LSD* and *MSD* comparing to *Quicksort* is larger when we use 64 bit keys instead of 32 bit keys. On the other hand *Binary Quicksort* and *Three-Way Radix Quicksort* are slower than standard *Quicksort*. The efficiency of *Binary Quicksort* rapidly gets lower when key values in sorted files repeat.

Reference

[1] Sedgewick, Robert. *Algorithms in C++, Parts 1-4, Fundamentals, Data structures, Sorting, Searching.* Addison-Wesley, 2004, ISBN: 0-201-35088-2.

[2] Sedgewick, Robert. *Algoritmy v C, Části 1-4, Základy, Datové struktury, Třídění, Vyhledávání.* Softpress, Praha, 2003, ISBN: 80-86497-56-9.

A. Obsah přiloženého DVD

K tomuto textu je přiloženo DVD s tímto obsahem:

`bin/`
: Složka obsahuje instalátor *RadixSortAlgorithmsInstall.exe* programu *RadixSortAlgorithms*. Ze složky `bin/RadixSortAlgorithms` je možné spustit program bez instalace přímo z DVD.

`doc/`
: Zde je uložen tento text ve formátu PDF - `Algoritmy cislicoveho trideni.pdf`, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru (v ZIP archivu) v programu LaTeX.

`src/`
: Kompletní zdrojové kódy programu *RadixSortAlgorithms* (v ZIP archivu).

`readme.txt`
: Instrukce pro instalaci a spuštění programu *RadixSortAlgorithms*, včetně požadavků pro jeho provoz, popis obsahu DVD.