POROVNÁNÍ ŘADICÍCH ALGORITMŮ

Vladimír Hlaváč
ČVUT FS, hlavac@fs.cvut.cz

Abstrakt: Práce obsahuje výsledky praktického testování několika řadicích algoritmů a možnosti porovnání jejich skutečné časové náročnosti s jejich asymptotickou složitostí. V první části je prezentováno porovnání náročnosti operace porovnání pro celá čísla a pro textové řetězce, které nejsou kódovány přímo v ASCII, ale v některé z rozšířených znakových sad.
Klíčová slova: Bubble sort, Selection sort, Insertion sort, Heap sort, Merge sort

1. Metodika porovnávání

Pro porovnání bylo vybráno několik nejčastěji používaných řadicích algoritmů, jejichž chování je všeobecně známé.

1.1 Měření skutečné časové náročnosti

Protože se jedná o modelové řešení, jsou pochopitelně řazena čísla. Při praktickém použití algoritmů jsou nejčastěji řazeny záznamy (nebo objekty) podle zvoleného klíče, který obvykle spočívá v porovnání dvou řetězců. Při jakékoli zvolené metodě je tedy záměna pořadí dvou vyhledaných záznamů provedena záměnou patřičných ukazatelů, nebo při sestavování indexového souboru (například u databází) záměnou dvou čísel v poli indexů. Protože u rodiny x86 instrukce XCHG neumožňuje adresovat dvě adresy v paměti, jsou na záměnu vyžadovány minimálně tři instrukce procesoru (typicky 2× MOV a 1× MOVSD). Přesto je tato časová náročnost zanedbatelná ve srovnání s porovnáním dvou klíčů. Pokud má řazení být obecné, je třeba volat funkci operačního systému pro porovnání ve zvolené znakové sadě (např. UTF-8). Časová náročnost takové operace je řádově větší. Z tohoto důvodu bývá jako měřítko náročnosti algoritmu měřen počet nutných porovnání, nikoli operací se záznamy v paměti.

Časová náročnost (skutečný čas běhu programu v sekundách) pro kódování UTF-8 nebyla v rámci testů zjištěna, v následující tabulce je porovnání časů, které jsou třeba k porovnání dvou celých čísel, dvou čísel v řádové čárce a dvou řetězců v sadě Windows CP1250, ovšem nikoli prostředky operačního systému, ale rychlou knihovnou pro textové funkce. Za tímto účelem byl připraven prázdný cyklus (s ohledem na optimalizaci se musí stále uvnitř cyklu provádět funkce, které kompilátor nemůže zjednodušit), který byl následně doplněn o požadovanou funkci. Všechna data jsou získána jako průměr z deseti pokusů:

Tab. 1. Výpočet skutečné časové náročnosti některých operací

	čas [s]	opakování	čistý čas na provedení funkce [ns]	poměrná hodnota
prázdný cyklus (hodnota je v dalších řádcích odečítána)	1.131	100000000	(11.31)	- - -
celočíselné porovnání	1.291	100000000	1.596	1
reálná čísla	1.312	100000000	1.814	1.137
text v kódování CP1250	12.97	100000	129710	81272

V tab. 1 je v prvním sloupečku skutečný čas pokusu, v druhém počet opakování v rámci pokusu, ve třetím pak podíl těchto hodnot. Jednojádrový Celeron na 1.8 GHz. Čas měřen prostředky operačního systému. Porovnání dvou řetězců je v tomto případě 81272× pomalejší, než porovnání dvou celých čísel.

1.2 Testovací data

Pro testovací účely byla přes výše uvedené používána náhodná čísla. Funkce porovnání dvou čísel byla oddělena a doplněna o počítání počtu volání funkce. Pro porovnání s asymptotickou složitostí byla generována pole o počtu 16, 256, 4096 a 65536 prvků.

Protože některé metody pracují hůře s daty, která nejsou náhodná, byla jako kontrolní vzorek použita data, vzniklá načtením souboru „explorer.exe“ z adresáře operačního systému tak, že byl popsán jako pole neznaménkovaných celých 16bitových čísel. Pro generování více testovacích sad byl soubor čten vždy z jiného místa. Pro jedno konkrétní načtení byly pro vytvořené pole pomocí MS Excel zjištěny četnosti získaných hodnot. Zatímco hodnota „0“ se v rámci pole 65536 hodnot vyskytla 3255×, hodnota „65535“ 1205×, pak 54699 čísel se nevyskytlo ani jednou a 6011 čísel právě jednou. Na grafech je zjevné, že na výsledky testů nemá tato skutečnost žádný vliv.

2. Testované metody

Jednotlivé metody řazení jsou nejjednodušeji popsány na webových stránkách Algoritmy.net [1] (česky). Rozbor pokročilejších metod lze najít v publikacích [2] a [3]. Kniha [2] je stále dohledatelná na internetu, původně byla volně šiřitelná, citace odpovídá tištěné verzi.

2.1 Bubble sort

Bublinkové řazení. Byla vybrána základní varianta, kdy rozsah vnitřního cyklu se opakovaně zmenšuje o jedničku.

2.2 Selection sort

Řazení výběrem. Z neseřazeného zbytku pole je opakovaně vybírán nejmenší prvek, který je pak zaměněn s prvním prvkem neseřazeného seznamu – seřazená část se tím zvětší o nově zařazený prvek.

2.3 Insert sort

Základní verze algoritmu řazení vkládáním vezme první z neseřazené části, nalezne pro něj místo mezi již seřazenými tím, že jej porovná postupně od začátku, až najde první větší. Ten a celý zbytek seřazené oblasti posune a na uvolněné místo vloží řazený prvek. Metodě by prospělo přepsání do strojového kódu, protože většina procesorů (resp. jak ARM, tak Intel) mají pro takový přesun instrukci, která optimálním způsobem spolupracuje s pamětí.

2.4 Insert sort s vkládáním půlením intervalu

Jedna z mnoha variant řazení vkládáním využívá faktu, že část dat je již seřazena a správné místo na vložení lze najít rychleji. Hledání správného místa tak má místo náročnosti O(n) jen O(log n), takže výsledná metoda patří do kategorie O(n.log n). Nevýhodou je, že pokud porovnávaný prvek je stejný, jako vkládaný, vložíme jej na toto místo. Ve výsledku se tím ale může poškodit předchozí seřazení podle jiného klíče (říkáme, že metoda není stabilní).

Metodu lze upravit tak, že nalezne konec úseku, kde jsou stejné prvky, a nově řazený vloží tam. Pak již ale nelze zaručit uvedenou časovou náročnost (metoda se vrací do kategorie $O(n^2)$).

2.5 Quick sort

Před deseti až dvaceti roky nejoblíbenější metoda řazení. Z pole vezmeme jeden prvek (označuje se „pivot“) a pak začneme pole prohledávat z obou konců. Zleva hledáme první, který je větší. Zprava první, který je menší. Tyto dva pak vyměníme. Použitý prvek pak umístíme na místo, kde se oba výběry sejdou. Tím máme dvě části pole, první, kde jsou všechny prvky menší než pivot a druhé, kde jsou větší. Speciální případ je pak již zařazený prvek (pivot). Pro to, aby se jednalo o metodu „rozděl a panuj“, „divide-and-conquer“, která přenáší rychlost složky algoritmu z O(n) na O(log n) stačí, aby rozdělené části byly v nějakém jiném poměru, než 1:(n–1), postačí i 1:10. Z toho také vyplývá neschopnost této metody seřadit již seřazený seznam, kdy má náročnost $O(n^2)$.

Aby tato časová náročnost byla alespoň nepravděpodobná, používá se většinou randomizovaná (znáhodněná) verze algoritmu, kdy počáteční prvek se z pole vybírá náhodně. Algoritmus pak řadí velmi rychle téměř vždy, ale protože problematický případ s opakovanou volbou nejmenšího či největšího prvku nelze zcela vyloučit, formálně stále spadá do kategorie $O(n^2)$.

Protože všechny algoritmy byly realizovány v Pascalu, byla pro toto testování zvolena varianta metody, se kterou přišla firma Borland inc. (v grafech je tato metoda také popsána jménem této firmy). Inovace spočívá v tom, že jako rozhodující prvek pro řazení (pivot) se vybere ten, který je uprostřed pole. Pokud se pak při hledání z jedné strany (první větší než pivot zleva, první menší než pivot zprava) nenajde požadovaný prvek, ale z druhé strany ano, pak se jediný nalezený zamění s pivotem. V tom případě se ale musí vyhledávání z této strany opakovat, takže výsledná náročnost algoritmu je formálně stále $O(n^2)$. Druhá nevýhoda této varianty je, že není stabilní (přesunem pivotu může být porušeno pořadí prvků se stejným klíčem). Mimochodem, popsaný jeden cyklus této metody představuje nejrychlejší možnost nalezení mediánu.

2.6 Heap sort

Klíčovým prvkem je zde hromada (na [1] označovaná jako halda), sloužící jako prioritní fronta. Hromada je zvláštní druh stromu, kde (protože řadíme od nejmenšího) má rodič vždy menší hodnotu než jeho dva potomci. Hromada se zapisuje do pole tím způsobem, že pokud je rodič na adrese $\boldsymbol{i}$, pak jeho potomci jsou na adresách $2\boldsymbol{i}$ a $2\boldsymbol{i}+1$. Prvky na hromadu přidáváme tak, že nový přidáme na konec pole, a pak jej porovnáme s rodičem (na adrese např. j div 2); pokud je menší, tyto dva prohodíme a operaci opakujeme.

Seřazené pole pak získáme postupným odebíráním z hromady. Základní varianta algoritmu říká, že po odebrání vezmeme poslední prvek, umístíme ho místo chybějícího (tím se hromada zmenšila o jednu) a hromadu opravíme [1]. Rychlejší metoda spočívá v posunu hromady tak, že místo chybějícího vezmeme menšího z potomků, a akci opakujeme, dokud se nedostaneme na konec hromady. Teprve zde poslední prvek hromady dáme na poslední uvolněné místo. Formálně je třeba i pak hromadu opravit, ale většinou dojde jen k porovnání, případně k minimálnímu počtu opravných prohození.

Výhodou Heap sortu je, že po vybudování hromady máme vždy nejmenší prvek k dispozici okamžitě. Tam, kde jde o nějakou formu paralelních algoritmů, nemusí druhý algoritmus čekat na uspořádání hromady, to může proběhnout na pořadí. Prosté vybrání prvního prvku je pak v kategorii O(1).

2.7 Merge sort

Řazení slučováním. Protože u Quick sortu byl problém, že pole se nám nemuselo dělit rovnoměrně, je zde postup otočen. Pole se rozdělí na jednotlivé prvky a ty se opakovaně slučují. V průběhu algoritmu slučování probíhá tak, že máme dva ukazatele (do obou slučovaných polí), tyto prvky porovnáváme a menší zapisujeme do výsledného pole. Důsledkem tedy je, že musíme mít druhé stejně velké pole, kam zapisujeme mezivýsledky. Obě pole můžeme samozřejmě střídat, takže prostorová náročnost je 2n. To je důvod, proč v dobách, kdy paměť byla drahá, nebyl tento algoritmus populární. Z druhé strany je stabilní a vykoná maximálně $O(n.\log_2 n)$ porovnání, což je i nejmenší možný počet porovnání u dat, o kterých nemáme předběžné informace (pokud bychom například věděli, že data jsou v nějakém rozsahu a nějak rozložena, můžeme pro zrychlení použít Bucket sort, česky např. přihrádkové řazení).

3. Výsledky

Program je napsán tak, že zvolenou metodu spustí 16×, 256×, 4096× a 65536×, a změřené počty porovnání zapíše do okna typu Memo, které může mít více řádek. Bylo doplněno tlačítko, které spustí každou metodu 5×, takže program může pracovat samostatně a data lze pak snadno zpracovat do grafu. Data pro jednotlivou metodu vypadají například následovně:

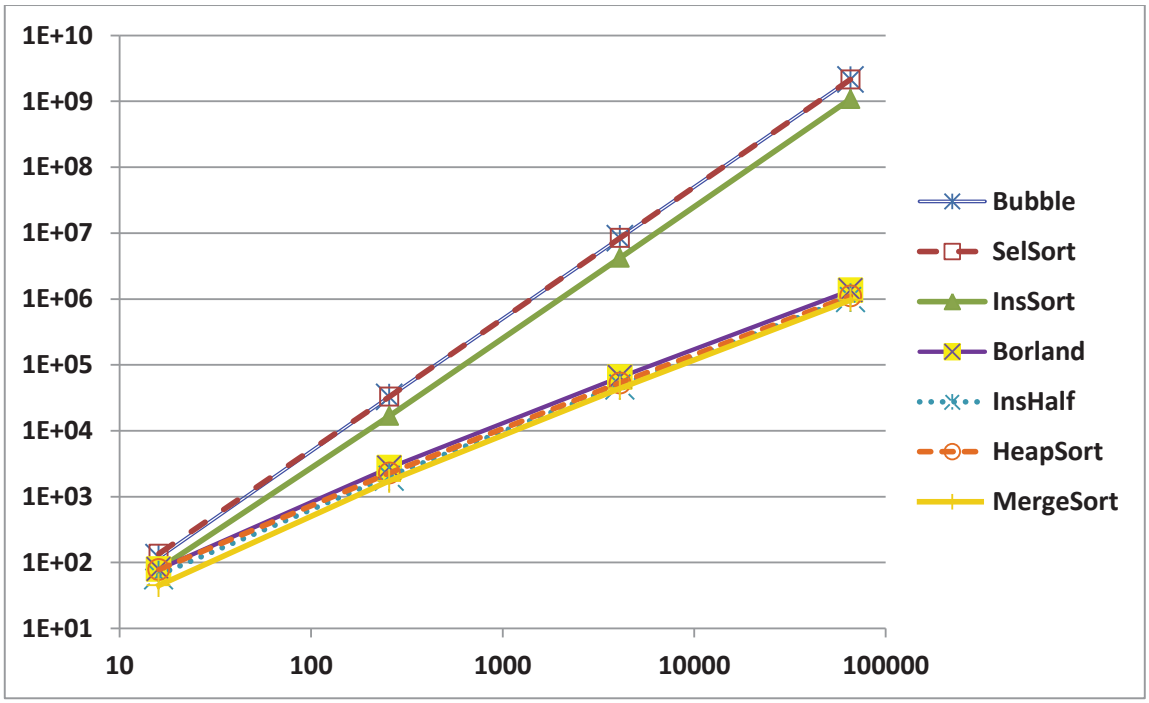
Tab. 2. Ukázka vygenerovaných dat pro jednu metodu. Umístění v řádcích je zavádějící, data v jednom řádku nemají společného nic jiného, než že vznikla při stejném spuštění.

Borland	69	2687	65320	1393786
Borland	78	2686	67526	1372598
Borland	76	2509	62157	1362484
Borland	90	3136	64287	1430948
Borland	82	2742	63409	1437918

Pro porovnání byla dávka spuštěna dvakrát pro náhodná data a dvakrát pro data získaná načtením souboru „explorer.exe“. Výsledné hodnoty byly zprůměrovány (deset od jedné metody) a vyneseny do grafu. Zde je třeba zdůraznit, že časová náročnost některých metod (tak, jak byly implementovány) nezávisí na datech, a místo průměru tak mohl být vzat první řádek. Jedná se o Bubble sort, Selection sort a variantu Insert sortu s vyhledáváním půlením intervalu.

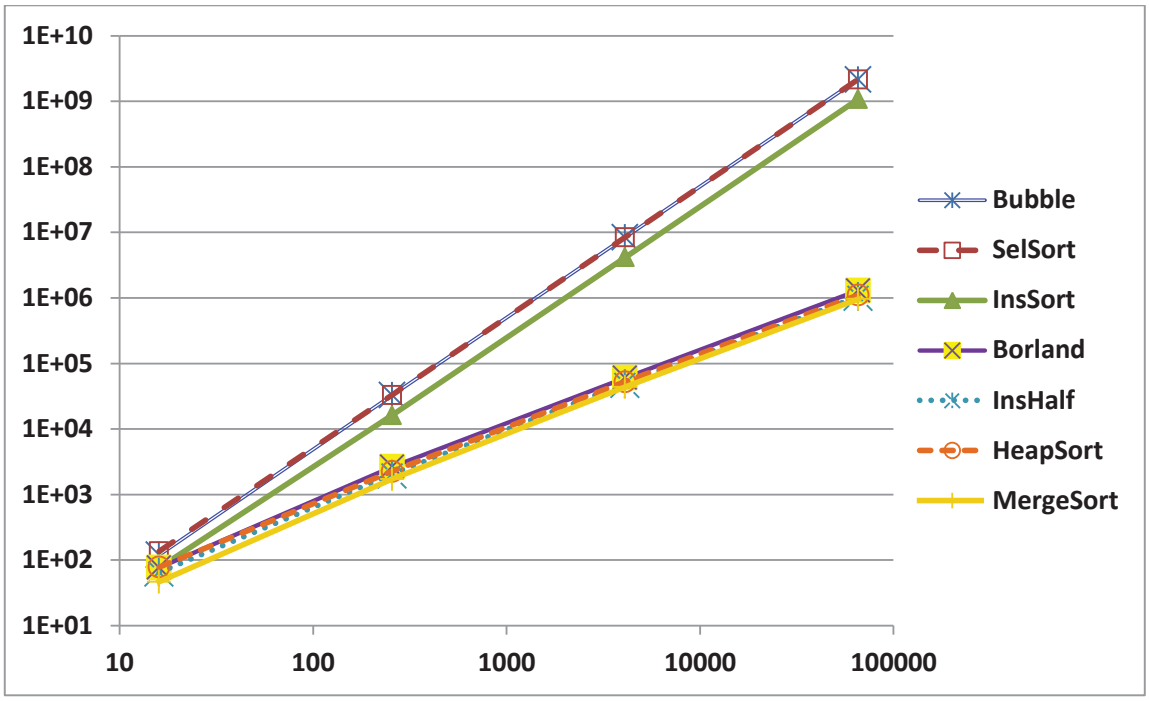
3.1 Souhrnný graf

Pro zobrazení v grafu byl použit graf se dvěma logaritmickými souřadnicemi, na vodorovné ose je počet prvků seřazovaného pole, na svislé je počet porovnání. Jsou použity dekadické logaritmy, logaritmus o základu 2 by umístil počty prvků i počty porovnání přehledněji.



Obr. 1. Odshora: Bublinkové řazení, řazení výběrem, řazení vkládáním, varianta Quick-sortu dodávaná s Turbo Pascalem firmy Borland, řazení vkládáním s půlením intervalu, řazení hromadou a slučováním.

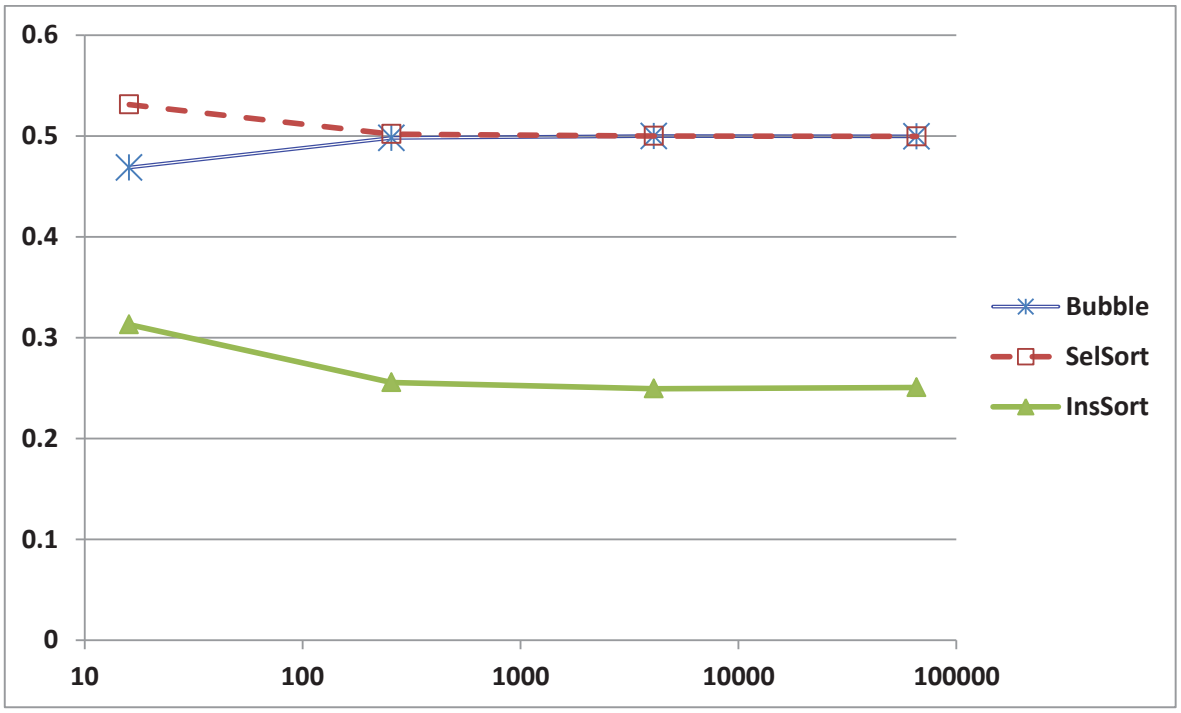
Tentýž graf pro druhý soubor, kde data nebyla náhodně rozdělena (četnosti), nicméně byla na začátku do značné míry náhodně seřazena.



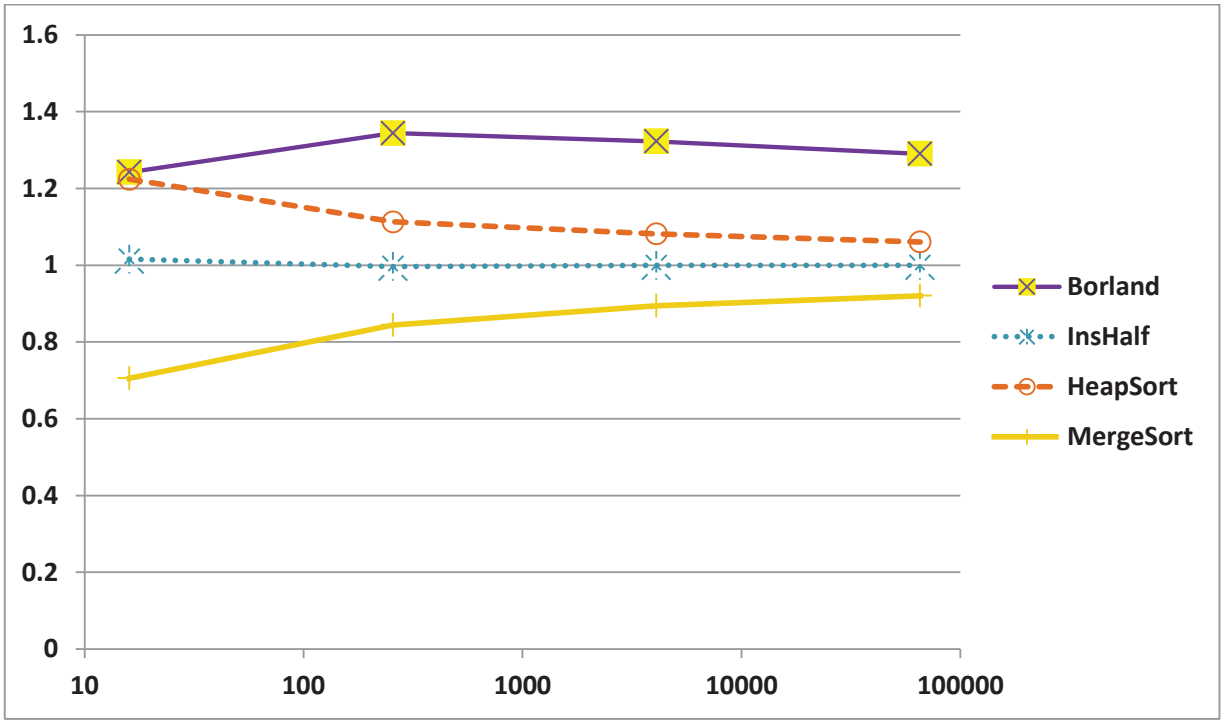
Obr. 2 Grafy pro data s jiným rozložením hodnot vypadají naprosto stejně. Jiné hodnoty by bylo možné získat, kdyby data byla předem v různé míře seřazena.

3.2 Graf pro porovnání s teoretickou náročností

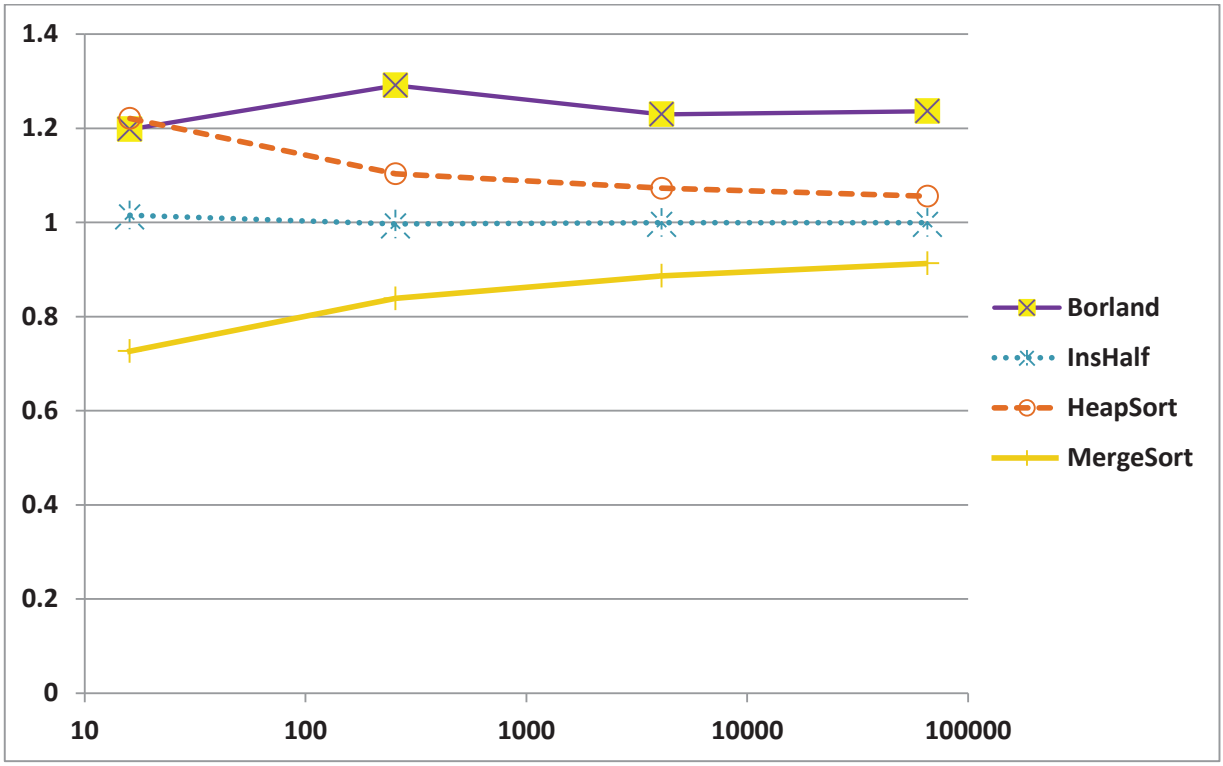
Přehlednější zobrazení výsledků bylo dosaženo tím, že jsou vyneseny jen zvolené grafy, a jejich skutečná časová náročnost je podělena jejich asymptotickou náročností (číslem n^2, resp. $n.\log_2 n$). Protože jde o očekávanou (nejpravděpodobnější) hodnotu, je Quick sort zařazen mezi algoritmy kategorie O(n.log n).



Obr. 3 Hodnoty náročnosti po podělení hodnotou n^2. U prvních dvou je výsledná hodnota přibližně jedna polovina, protože oba algoritmy procházejí oblast o délce postupně od n do 1. U řazení vkládáním je nejen průměrná délka procházené části již seřazených dat poloviční, ale i samotná seřazená část má na začátku nulovou délku, takže ve výsledku se časová náročnost blíží jedné čtvrtině.



Obr. 4 Quick sort (první graf) do uvedené kategorie vlastně ani nepatří, takže i když byl očekáván lepší výsledek, jedná se o úspěch. Řazení hromadou je obecně pomalejší, protože nejprve je třeba složit hromadu, a všechno se tak dělá vlastně dvakrát. Modifikace řazení vkládáním má pevnou časovou délku, modifikací algoritmu by ji šlo dokonce zkrátit, ale je nutno připomenout, že metoda není stabilní. Řazení slučováním (Merge sort) šetří čas při dořazování zbytku druhého pole – pokud se při slučování první jedno z polí vyčerpá, pak se druhé kopíruje bez porovnání. Speciální případ je tedy první krok, kdy se porovnávají jednotlivé prvky, a na každé dva připadne jen jedno porovnání. Odpovídá to tomu, že se jedná o dodnes nejrychlejší známou metodu řazení (bez požadavku na předběžnou znalost vlastností dat).



Obr. 5 Tytéž průběhy pro data získaná načtením souboru „explorer.exe“. Nerovnoměrnost dat má vliv jen na Quick sort. Kdyby bylo pořadí dat předem částečně seřazené, projevil by se vliv i na Merge sort, ale zde se nejedná o tento případ.

4. Závěr

Při porovnání algoritmů kategorie n^2 byla získána očekávaná data. Z těchto algoritmů má jistý smysl Bubble sort, protože jej lze napsat rychleji, než se dohledá knihovna řadicích algoritmů na internetu, a je také velmi spolehlivý. Při posuzování časové náročnosti programu je třeba sečíst celkovou dobu, kdy jej budou uživatelé používat (krát hodinová cena práce operátorů) a čas, který s psaním aplikace stráví programátor (krát jeho hodinová sazba). Pokud program nebude prakticky používán, je zbytečné jej optimalizovat.

Při porovnání výkonnějších algoritmů překvapila nízká výkonnost Quicksortu v porovnání s Heapsortem. Opět tu ale platí jednoduchost implementace, která má přímou vazbu na spolehlivost programu.

Pro vytvoření testovacích programů bylo použito prostředí Lazarus. Nebylo možné otestovat přímo UTF-8, který je svou proměnnou délkou znaků velmi problematický, protože tato znaková sada nebyla dostupná v kombinaci se starší verzí Windows (není přímá cesta k načtení dat). Nativní podporu mají textové soubory v UTF-8 v Lazarusu jen v případě, že je provozován v prostředí Linux.

Literatura

[1] Neckář, Jan: Algoritmy.net. Příklady algoritmů v jazyce Java, Perl, Python, řešení složitých matematických úloh Webová stránka, online: http://algoritmy.net
[2] Dasgupta, Sanjoy, Papadimitriou, Christos H.,Vazirani, Umesh V.: Algorithms. McGraw-Hill Education – Europe, 2006.
[3] Erickson, Jeff: Algorithms. 2015. Dostupné on-line: http://www.cs.illinois.edu/~jeffe/teaching/algorithms/ .

Selected article from
Tento dokument byl publikován ve sborníku

Nové metody a postupy v oblasti přístrojové techniky, automatického řízení a informatiky 2019

New Methods and Practices in the Instrumentation, Automatic Control and Informatics 2019

27. 5. – 29. 5. 2019, Zvíkovské Podhradí

ISBN 978-80-01-06617-1

Web page of the original document:
http://iat.fs.cvut.cz/nmp/2019.pdf

Obsah čísla/individual articles:
http://iat.fs.cvut.cz/nmp/2019/

Ústav přístrojové a řídicí techniky, FS ČVUT v Praze, Technická 4, Praha 6