EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/B/2018/36097107837859588

Porovnanie exekučnej efektívnosti triediacich algoritmov Shell Sort a Insertion Sort usporadúvajúcich veľké súbory dát v programe vytvorenom v jazyku C

Bakalárska práca

2018 Peter Belko

EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Porovnanie exekučnej efektívnosti triediacich algoritmov Shell Sort a Insertion Sort usporadúvajúcich veľké súbory dát v programe vytvorenom v jazyku C

Bakalárska práca

Študijný program: Hospodárska informatika

Študijný odbor: 9.2.10 Hospodárska informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2018 Peter Belko

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú prácu vypracoval samostatne a že som uviedol všetku použitú literatúru súvisiacu so zameraním bakalárskej práce.

Dátum: Podpis študenta:

Poďakovanie

Srdečne by som chcel poďakovať všetkým, ktorí mi pomáhali počas písania bakalárskej práce. Predovšetkým ďakujem môjmu školiteľovi Ing. Igorovi Košťálovi, PhD. za ochotný prístup, pomoc pri písaní práce a v neposlednom rade za tolerantnosť.

36097107837859588

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Peter Belko

Študijný program: hospodárska informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)

Študijný odbor: 9.2.10 Hospodárska informatika

Typ záverečnej práce: Bakalárska záverečná práca

Jazyk záverečnej práce: slovenský

Sekundárny jazyk: anglický

Názov: Porovnanie exekučnej efektívnosti triediacich algoritmov Shell Sort a Insertion Sort usporadúvajúcich veľké súbory dát v programe vytvorenom v jazyku C

Anotácia: Študent v práci zanalyzuje rôzne druhy triediacich algoritmov, ich princípy, použitie a ich implementáciu v programe vytvorenom v jazyku C s detailnejším zameraním sa na triediace algoritmy Shell Sort a Insertion sort. V rámci bakalárskej práce študent vytvorí program v jazyku C, v ktorom implementuje oba triediace algoritmy Shell Sort a Insertion Sort. Vytvorený

program bude schopný zmerať exekučnú efektívnosť oboch triediacich algoritmov pri usporadúvaní veľkého súboru dát, napr. poľa so stotisíc prvkami. Pomocou výstupov programu študent vykoná porovnanie exekučnej efektívnosti triediacich algoritmov Shell Sort a Insertion Sort pre veľký súbor dát a vyhodnotí toto porovnanie.

Vedúci: Ing. Igor Košťál, PhD.

Katedra: KAI FHI - Katedra aplikovanej informatiky FHI

Vedúci katedry: Ing. Mgr. Peter Schmidt, PhD.

Dátum zadania: 19.04.2017

Dátum schválenia: 22.05.2018 Ing. Mgr. Peter Schmidt, PhD.

vedúci katedry

Abstrakt

BELKO, Peter: Porovnanie exekučnej efektívnosti triediacich algoritmov Shell Sort a Insertion Sort usporadúvajúcich veľké súbory dát v programe vytvorenom v jazyku C – Ekonomická univerzita v Bratislave, Fakulta hospodárskej informatiky, Katedra aplikovanej informatiky. Vedúci záverečnej práce Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2017, 50 strán.

Cieľom bakalárskej práce je vypracovanie programu v programovacom jazyku C, ktorý porovnáva exekučnú efektívnosť triediacich algoritmov Insertion Sort a Shell Sort. V teoretickej časti práce boli postupne opísané základné triediace algoritmy, ich časová a pamäťová náročnosť. Pri každom algoritme sú taktiež spomenuté jeho najväčšie výhody a vizualizácia postupnosti krokov daného algoritmu. Praktická časť bakalárskej práce sa zameriava na vytvorenie a popis programu, ktorý porovnáva exekučnú efektívnosť vybraných triediacich algoritmov a vyhodnotenie výsledkov meraní časovej náročnosti. Oba triediace algoritmy triedili rovnaké číselné pole, ktoré bolo naplnené náhodne generovanými číslami. Veľkosť a rozsah náhodne generovaných čísiel si zadefinoval používateľ pri spustení programu. Používateľ si takisto zadefinoval aj počet jednotlivých testovaní pre určený počet prvkov poľa. Program taktiež zmeral čas každého zoradenia prvkov v poli a skontroloval, či sú tieto polia zoradené v správnom poradí. V závere je zhodnotené, ktorý algoritmus je efektívnejší podľa výsledkov priemerných časov z testovaní.

Kľúčové slova:

triediaci algoritmus, exekučná efektívnosť, číselné pole, shell sort, insertion sort

Abstract

BELKO, Peter: A comparison of an execution efficiency of Shell Sort and Insertion Sort sorting algorithms that sort large data sets in a program created in C – University of Economics in Bratislave, Faculty of Economic Informatics, Department of Applied Informatics. Supervisor of final thesis Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2017, 50 pages.

The aim of this bachelor thesis is to create a program in the programming language C, which compares the effectiveness of the sorting algorithms Insertion Sort and Shell Sort. In th theoretical part of the thesis the basic sorting algorithms were described along with their time and memory complexity. With each algorithm, their advantages and visualization of the sequence of steps are shown. The practical part of the bachelor thesis focues on the creation and description of the program, which compares the effectiveness of the selected sorting algorithms and the evaluation of the results of the time intensity measurements. Both sorting algorithms sorted the same numeric field that was filled with randomly generated numbers. The size and range of randomly generated numbers is defined by the user when the program is started. The user has also defined the number of individual tests for the specified number of fields elements. The program also measured the time of each sort of elements in the array and checked whether these field were sorted in the correct order. In conclusion, it is evaluated which algorithm is more effective based on the results of average test times.

Keywords:

sorting algorithm, execution efficiency, numeric array, shell sort, insertion sort

Obsah

Úvod

Usporadúvanie prvkov je aj v bežnom živote veľmi potrebné a odporúčané, pretože pomáha dáta organizovať tak, aby bola ďalšia práca s dátami menej náročná a viac efektívna. Typickými príkladmi triedenia je napríklad usporiadanie slov v slovníku podľa abecedy alebo zoradenie detí podľa výšky.

Podobne je to aj v informatike, kedy je usporiadané pole vhodnejšie na ďalšie spracovanie a algoritmy, ktoré pracujú s utriedeným poľom dosahujú lepší exekučný čas. Súčasné problémy v informatike sú prevažne komplexné a ich riešenie musí byť efektívne, a preto je dáta vhodné najskôr utriediť.

Triediaci algoritmus je algoritmus, ktorý ukladá prvky poľa v špecifickej postupnosti. Efektívne triedenie je dôležité pre optimálne použitie ďalších algoritmov, ako napríklad algoritmy vyhľadávania, ktoré očakávajú na vstupe práve utriedené dáta. Taktiež sú utriedené dáta pre ľudí viac zrozumiteľné.

Každý algoritmus je svojim spôsobom špecifický, či už časovou alebo pamäťovou zložitosťou. Každý triediaci algoritmus má svoje výhody a svoje nevýhody. [1]

1 Súčasný stav riešenej problematiky doma a v zahraničí

Literatúra porovnáva triediace algoritmy z rôznych uhlov pohľadov, pričom sa zameriavajú na rôzne výhody a nevýhody daných algoritmov. Táto práca je zameraná na vytvorenie programu v programovacom jazyku C, ktorý porovná exekučný čas triediacich algoritmov Insertion Sort a Shell Sort usporadúvajúcich veľké súbory dát.

V súčasnosti sú tieto algoritmy dostupné, odbornou verejnosťou dobre známe a veľmi často používané. Z týchto dôvodov je súčasný stav na Slovensku podobný ako je stav v zahraničí.

1.1 Triedenie

Triedenie je proces, pri ktorom sa mení poradie prvkov tak, že sa ich snažíme usporiadať do nejakého logického poradia. V informatike je triedenie algoritmus, ktorý zoraďuje prvky zoznamu v určenom poradí. Medzi najpoužívanejšie poradie patria numerické a lexikografické poradie.

Výstup triediacich algoritmov by mal spĺňať dve podmienky:

Výstup je vo vzostupnom poradí (každý prvok je väčší ako predchádzajúci prvok)

Výstup je permutácia vstupu (výstup musí byť tvorený zo všetkých vstupných prvkov)

Od počiatkov informatiky priťahoval problém triedenia veľký záujem výskumníkov, snáď vďaka zložitosti efektívneho riešenia napriek jednoduchému a intuitívnemu zadaniu. Napríklad Bubble Sort bol analyzovaný už v roku 1956 [2]. Hoci mnohí triedenie považujú za vyriešený problém, ešte aj dnes sa stále vyvíjajú užitočné nové triediace algoritmy (napríklad Library Sort bol prvýkrát publikovaný v roku 2004).

Triediace algoritmy sú prevažne prezentované v úvodných kurzoch informatiky a programovania, kde hojnosť algoritmov riešiacich jeden problém poskytuje jemný úvod k rôznym fundamentálnym konceptom algoritmizácie ako notácia veľké O, algoritmy rozdeľ a panuj, údajové štruktúry, probabilistické algoritmy a analýza zložitosti. [3]

1.2 Klasifikácia triediacich algoritmov

Triediace algoritmy je možné klasifikovať podľa rôznych kritérií. Toto delenie napomáha lepšiemu pochopeniu konkrétneho algoritmu, a teda pomáha aj lepšiemu výberu z určitej skupiny pre vyriešenie konkrétneho problému. Ide nielen o výpočtovú zložitosť, využitie pamäte, ale aj o stabilitu programu, či spôsob, akým algoritmus vykonáva triedenie.

Výpočtová zložitosť triediacich algoritmov

Výpočtová zložitosť alebo výpočtová náročnosť je pojem z teórie algoritmov, vyjadruje nakoľko je výpočet podľa zvoleného algoritmu zložitý. Výpočtovú zložitosť študuje teória zložitosti. Závislosť, ktorá vyjadruje skutočný počet operácií, potrebných na realizáciu algoritmu, ako funkcia, závisiaca od vstupných údajov, sa nazýva časová výpočtová zložitosť.

Výpočtová zložitosť má dve základné miery:

- časová zložitosť
- pamäťová zložitosť

Optimalizácia algoritmu sa týka minimalizácie jednej alebo obidvoch mier zložitosti.

Ďalšou veľmi dôležitou charakteristikou algoritmov usporadúvania je stabilita. Algoritmus je stabilný, ak zachováva relatívnu postupnosť rovnakých prvkov množiny. To znamená, že ak usporiadame abecedne usporiadaný zoznam študentov podľa ich prospechu, potom stabilný algoritmus abecedne usporiada študentov s rovnakým prospechom, ale nestabilný algoritmus je náchylný produkovať usporiadanie bez ohľadu na predošlé usporiadanie. [4]

2 Prehľad triediacich algoritmov

V nasledujúcej časti budú predstavené základné triediace algoritmy, spôsob ich fungovania, ich časová náročnosť a taktiež pamäťová náročnosť. Ku každému algoritmu je priložený pseudo kód a vizualizácia priebehu triedenia pre ľahšie pochopenie daného algoritmu.

2.1 Triedenie vkladaním

Základný princíp fungovania je, že prvky sú posudzované postupne po jednom a každý nový prvok je vložený na vhodnú pozíciu, ktorá je relatívna k predošlému poradiu prvkov.

2.1.1 Insertion Sort

Opis algoritmu

Insertion Sort je jedným zo základných typov triediacich algoritmov, ktorý funguje na princípe rozdelenia triedeného poľa na dve časti – utriedená časť a neutriedená časť. Utriedená časť predstavuje časť poľa, kde je postupnosť prvkov usporiadaná vzostupne a to znamená, že žiadny nasledujúci prvok nie je menší ako predchádzajúci prvok. Druhá časť poľa – neutriedená časť, obsahuje prvky, ktoré zatiaľ neboli utriedené a teda stále môže obsahovať prvky v nesprávnom poradí.

Postup akým Insertion Sort triedi je nasledujúci.

Z celého neutriedeného poľa sa zoberie prvý prvok a označí sa za usporiadaný. Potom sa zoberie nasledujúci prvok a zaradí sa do usporiadanej časti poľa na správne miesto. Takto sa prejdú všetky prvky poľa až do konca, a tým pádom môžeme na konci povedať, že je pole usporiadané.

Časová zložitosť

V najhoršom prípade: $O(n^2)$ – tento prípad nastáva práve vtedy, keď je pole usporiadané naopak

V najlepšom prípade: $\Omega(n)$ – tento prípad nastáva, keď je pole už v usporiadanom tvare a algoritmus musí prejsť celým polom aspoň raz, aby to overil

Priemerný prípad: $\Theta(n^2)$

Pamäťová zložitosť

Ako výhoda algoritmu Insertion Sort je, že nevyžaduje skoro žiadnu dodatočnú pamäť. Algoritmus využíva najmä pamäť vstupného poľa a konštantnú veľkosť pamäte, pomocou ktorej je realizované vymenenie prvkov.

Výhody a využitie

Veľkými výhodami algoritmu Insertion Sort sú jeho jednoduchosť a efektívnosť. Pri porovnaní s inými algoritmami je Insertion Sort približne dvakrát rýchlejší než Bubble Sort a približne o 40% rýchlejší než Selection Sort.

Avšak pri usporadúvaní postupností s veľkým počtom prvkov je Insertion Sort neefektívny. [5]

Pseudo kód triediaceho algoritmu Insertion Sort

procedúra insertion_sort(IT = nezotriedené prvky v poli) {

pre každý prvok p v poli IT {

porovnávaj p s prvkami naľavo od neho kým nenarazíme na menší prvok než p,

posunieme porovnávaný prvok o jedno doprava potom na miesto prvého väčšieho prvku než p zapíšme p

}

}[7]

Vizualizácia postupu triedenia algoritmom Insertion Sort

Utriedenie nasledujúceho číselného poľa algoritmom Insertion Sort.

Insertion Sort začne porovnávaním prvých dvoch prvkov.

Zistí, že oba prvky 14 a 33 sú už v správnom poradí. Na teraz je však iba 14 považovaná za utriedenú časť poľa.

Insertion Sort sa posunie a porovnáva 33 s 27.

Zistí, že 27 je menšie, čiže 33 nie je na správnom mieste.

Vymení 33 za 27. Zároveň, ale taktiež prvok 27 porovná postupne so všetkými prvkami utriedenej časti poľa. V tomto prípade je možné vidieť, že sa v tejto časti nachádza len jeden prvok – 14. Prvok 27 je väčší ako 14, a preto je 27 umiestnená za 14. Utriedená časť poľa sa zväčšila o jeden prvok.

Ako ďalšie nasleduje porovnanie 33 a 10.

Prvky nie sú v utriedenom poradí.

Preto sú vymenené.

Avšak ani takto by utriedená časť nebola správna, pretože 27 je väčšia ako 10.

A preto sú vymenené aj tieto prvky.

A znova sú prvky 14 a 10 v neutriedenom poradí.

Aj tieto prvky sú vymenené. Na konci tretej iterácie sa teda v utriedenej časti nachádzajú 3 prvky.

Takto algoritmus pokračuje až kým neutriedi všetky prvky z neutriedenej časti do utriedenej časti, a teda až kým nebude celé pole utriedené.

2.1.2 Shell Sort

Opis algoritmu

Shell Sort je jedným zo základných typov triediacich algoritmov, ktorý funguje na podobnom princípe ako Insertion Sort, avšak je vylepšený tým, že sa najskôr snaží zoradiť veľké čísla na pravú stranu a malé čísla na ľavú stranu. Tento algoritmus je efektívnejší oproti algoritmu Insertion Sort, v ktorom prebiehajú výmeny iba medzi susednými prvkami. Ak sa napríklad stane, že prvok s najmenšou hodnotou je na konci poľa, tak na to, aby sa dostal kam patrí, treba toľko krokov, koľko je počet prvkov poľa. Shell Sort je jednoduchým rozšírením Insertion Sort, ktorý je efektívnejší tým, že umožní zámenu vzdialenejších prvkov.

Časová zložitosť

V najhoršom prípade: $O(n^2)$ – najhorší prípad nastáva, keď je zvolená najhoršia medzera

V najlepšom prípade: $\Omega(n \log n)$ – tento prípad nastáva práve vtedy, keď je pole usporiadané naopak

Priemerný prípad: $\Theta(n \log^2 n)$

Pamäťová zložitosť

Tak isto ako Insertion Sort, aj tento algoritmus nevyžaduje skoro žiadnu dodatočnú pamäť. Algoritmus využíva najmä pamäť vstupného poľa a konštantnú veľkosť pamäte, pomocou ktorej je realizované vymenenie prvkov.

Výhody a využitie

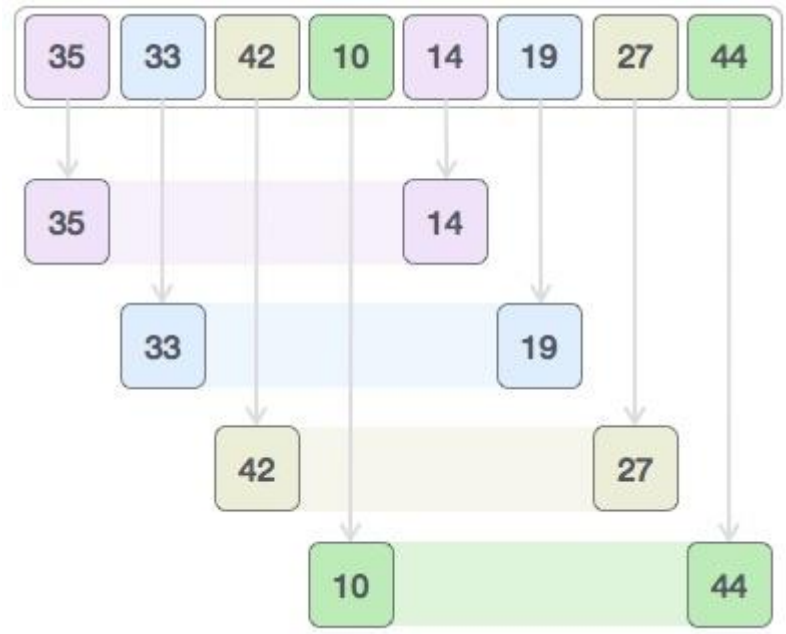
Shell Sort je najrýchlejší algoritmus zo všetkých spomínaných algoritmov patriacich do skupiny zložitosti $O(n^2)$. Pokiaľ porovnávame Shell Sort s jeho konkurentmi v tejto skupine, je viac ako päťkrát rýchlejší než Bubble Sort a dvakrát rýchlejší než Insertion Sort. Spomalenie metódy sa prejavuje až pri usporadúvaní postupnosti s viac ako 5000 prvkami. [6]

Pseudo kód triediaceho algoritmu Shell Sort

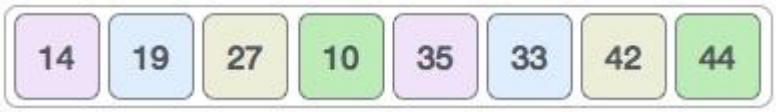
```
procedúra shell_sort(IT = nezotriedené pole prvkov)  {
        nastav medzeru
        rob, kým je medzera väčšia ako 0 {
            pre každú dvojicu prvkov medzi ktorými je medzera {
                ak sú dva prvky z dvojice v zlom poradí     {
                    vymeň ich
                }
            }
            zmenši medzeru na polovicu
        }
}
```
[7]

Vizualizácia postupu triedenia algoritmom Shell Sort

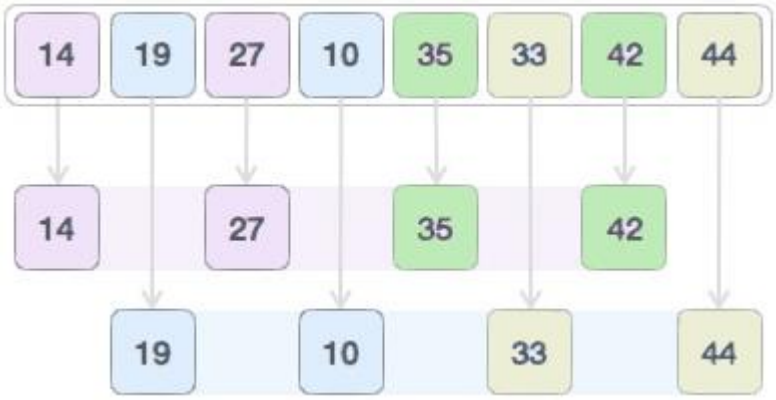
Nasledujúci príklad vysvetlí princíp fungovania algoritmu Shell Sort. Medzera je nastavená na štyri pozície. Podľa tejto medzery sa vytvoria dvojice prvkov, ktoré budú navzájom porovnávané. V tomto prípade sú tieto hodnoty: {35, 14}, {33, 19}, {42, 27} a {10, 44}.



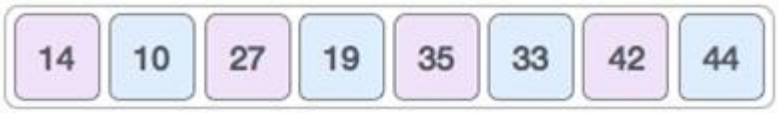
Ako ďalšie sa postupne prechádza cez všetky tieto dvojice a porovnávajú sa v nich jednotlivé prvky a v prípade, že je prvý prvok väčší ako druhý, navzájom sa vymenia ich pozície. Po tomto kroku by pole malo vyzerať takto -



Potom sa medzera zmenší o polovicu, čiže z pôvodných 4 sa zmení medzera na 2.



Tieto dvojice sú porovnávané a ak treba, pozície sa navzájom vymenia. Po tomto kroku by pole malo vyzerať nasledovne -



Ako posledný krok je pole triedené s medzerou nastavenou na 1. Shell Sort využíva princíp algoritmu Insertion Sort, kde je neutriedený prvok umiestnený na správnu pozíciu.

Utriedenie by vyzeralo nasledovne -

Ako môžeme vidieť, na utriedenie boli potrebné už len 4 výmeny.

2.2 Triedenie výmenou

Základný princíp fungovania je, že algoritmus kontroluje postupne dvojice susediacich prvkov a vymení tie, ktoré sú v nesprávnom poradí.

2.2.1 Bubble Sort

Opis algoritmu

Bubble Sort je jedným z najjednoduchších triediacich algoritmov na implementáciu. Algoritmus, ako opisuje názov, vystihuje chovanie sa vzdušných bubliniek, kedy malé (ľahké) bublinky stúpajú hore podobne ako sa malé prvky v poli posúvajú dopredu.

Bubble Sort je založený na opakovanom prechádzaní poľa, kedy porovnáva dva susedné prvky a v prípade, že je dvojica prvkov v zlom poradí, vymení ich pozíciu. Takto algoritmus prechádza celé pole, až kým nie je vykonaná žiadna výmena a vtedy Bubble Sort skončí, pretože sa s istotou dá povedať, že všetky prvky poľa sú usporiadané.

Časová zložitosť

V najhoršom prípade: $O(n^2)$ – tento prípad pre Bubble Sort nastáva, keď je pole usporiadané presne naopak. Vtedy je na utriedenie každého prvku potrebný vždy maximálny počet krokov.

V najlepšom prípade: $\Omega(n)$ – tento prípad nastáva, keď sú všetky prvky v poli usporiadané a algoritmus musí teda iba raz prejsť celým polom, aby skontroloval, či nie je niečo čo treba vymeniť

Priemerný prípad: $\Theta(n^2)$

Pamäťová zložitosť

Ako výhoda algoritmu Bubble Sort je, že nevyžaduje skoro žiadnu dodatočnú pamäť. Algoritmus využíva najmä pamäť vstupného poľa a konštantnú veľkosť pamäte na vymenenie prvkov.

Výhody a využitie

Ako aj pri ostatných algoritmoch, ktoré majú časovú zložitosť $O(n^2)$, nie sú tieto algoritmy určené na usporadúvanie veľkých počtov prvkov, pretože ich exekučný čas rastie na základe počtu prvkov veľmi rýchlo.

Avšak pri skoro utriedených poliach, t. j. také polia, ktoré majú neutriedených iba pár prvkov, respektíve keď treba vymeniť iba susedné prvky, je exekučný čas pomerne lepší.

Výhodou je taktiež ľahká implementácia a tým pádom je vhodný ako vyučujúci príklad teórie algoritmov a triediacich algoritmov.

Pseudo kód triediaceho algoritmu Bubble Sort

```
procedúra bubble_sort(IT = nezotriedené prvky v poli)  {
        logická premenná výmena;
        rob   {
                nastav výmenu na nepravdu   pre všetky indexy poľa IT okrem nuly    {
                        ak sú susedné dva prvky v zlom poradí     {
                                vymeň ich
                                nastav príznak výmeny na pravdu
                        }
                }
        } dokým sa prvky aspoň raz vymenili
}
```
[7]

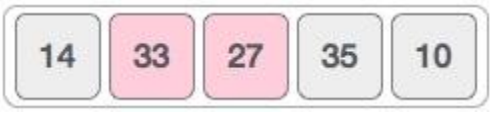
Vizualizácia postupu triedenia algoritmom Bubble Sort

Utriedenie nasledujúceho číselného poľa algoritmom Bubble Sort.

Bubble Sort začína na prvých dvoch prvkoch, ktoré porovná, aby zistil, ktorý je väčší.

V tomto prípade je číslo 33 väčšie ako 14, takže tieto dva prvky už sú v správnom poradí. Ako ďalšie Bubble Sort porovnáva čísla 33 a 27.

Nakoľko je 27 menšie ako 33, tieto dve hodnoty musia byť vymenené.



A tým pádom nové pole vyzerá takto –

Ako ďalšie sa porovnávajú čísla 33 a 35, ktoré sú taktiež v správnom poradí.

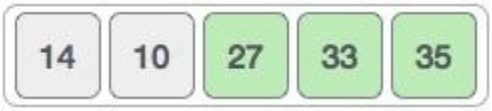
Bubble Sort sa posunie na nasledujúce čísla, ktoré sú 35 a 10.

Vieme, že 10 je menšie ako 35 a preto sú tieto dve čísla vymenené.

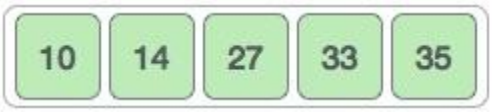
Po vymenení týchto čísiel sa dostávame na koniec poľa a po prvej iterácii by malo pole vyzerať takto –

Po ďalšej iterácii by pole vyzeralo nasledovne –

Je možné si všimnúť, ako sa po každej iterácii zoskupujú najväčšie prvky na konci poľa



Až kým počas iterácie netreba žiadnu výmenu, a tým pádom je pole usporiadané.



2.3 Triedenie vyberaním

Základným princípom triedenia vyberaním je, že najskôr je nájdený najmenší (resp. najväčší) prvok, potom ho istým spôsobom oddelíme od zvyšku. Následne sa proces opakuje na zvyšných prvkoch množiny.

2.3.1 Selection Sort

Opis algoritmu

Selection Sort je taktiež jedným z najviac prirodzených algoritmov na ľudské pochopenie. Selection Sort funguje na princípe vyberania minimálneho alebo maximálneho prvku z neusporiadanej časti poľa.

Algoritmus najprv v prvej iterácii vyberie najmenší prvok a umiestni ho na prvú pozíciu poľa. Túto časť označí za usporiadanú a neusporiadané pole teda skráti o jeden prvok. V druhej iterácii algoritmus opäť hľadá najmenší prvok z neutriedenej časti a po jeho nájdení ho umiestni na druhé miesto usporiadanej časti poľa. Takto pokračuje, až kým nie sú všetky prvky usporiadané.

Takto utriedené pole je utriedené vzostupne, t. j. od najmenšieho po najväčšie. To sa dá ľahko zmeniť na zostupné usporiadanie buď tým, že budeme vyberať namiesto minimálneho prvku maximálny prvok alebo budeme minimálny prvok zoraďovať na konci postupnosti.

Časová zložitosť

V najhoršom prípade: $O(n^2)$ – na nájdenie najmenšieho prvku z neusporiadanej časti poľa je vždy potrebné skontrolovať každý prvok

V najlepšom prípade: $\Omega(n^2)$ – tak isto ako v najhoršom prípade; na nájdenie minimálneho prvku je potrebné skontrolovať všetky prvky

Priemerný prípad: $\Theta(n^2)$

Pamäťová zložitosť

Tak isto ako Bubble Sort, výhodou algoritmu Selection Sort je, že nevyžaduje skoro žiadnu dodatočnú pamäť. Algoritmus využíva najmä pamäť vstupného poľa a konštantnú veľkosť pamäte na vymenenie prvkov ako pomocnú pamäť.

Výhody a využitie

Najväčšia výhoda Selection Sort spočíva v jeho jednoduchosti. Taktiež sa využíva najmä na vyučovanie základov tvorby algoritmov a ako vhodný príklad na vyučovanie triediacich algoritmov. Jeho najväčšia výhoda je práve jeho pamäťová zložitosť, kde algoritmus nevyžaduje od systému veľkú dodatočnú pamäť.

Pseudo kód triediaceho algoritmu Selection Sort

```
procedúra selection_sort(IT = nezotriedené prvky v poli)  {
        celé číslo i pre i idúce od 0 až po veľkosť IT - 1   {
                prvok p = najmenší prvok v IT v rozmedzí [i .. veľkosť IT]
                vymeň prvok v IT na pozícii i s prvkom p
        }
}
```
[7]

Vizualizácia postupu triedenia algoritmom Selection Sort

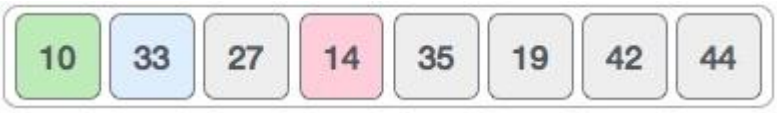
Utriedenie nasledujúceho číselného poľa algoritmom Selection Sort.

Na obsadenie prvej pozície v utriedenej časti poľa je najskôr skontrolované celé pole, pričom sa hľadá najmenší prvok. Na tejto pozícii sa najskôr nachádza 14, avšak ako najmenší prvok poľa bola nájdená hodnota 10.

Na základe toho bola vymenená 14 s hodnotou 10, a teda po prvej iterácii sme umiestnili hodnotu 10 na prvé miesto v utriedenej časti poľa.

Pre obsadenie druhej pozície, kde sa na tejto pozícii najskôr nachádza 33, bolo potrebné prehľadať zvyšnú časť neutriedenej časti poľa a nájsť z nej najmenší prvok.

Ako najmenší prvok bola nájdená hodnota 14 a bola vymenená s hodnotou 33.



Po druhej iterácii sú utriedené dva najmenšie prvky.

Podobne sa proces aplikuje aj na zvyšok poľa –

Po utriedení posledného prvku je celé pole usporiadané.

2.4 Triedenie spájaním

2.4.1 Merge Sort

Opis algoritmu

Merge Sort vo svojej funkcionalite využíva princíp rozdeľ a panuj, kde postupne pole najskôr rozdelí na dve rovnaké časti, a potom aj tieto dve časti opäť rekurzívne rozdelí na ďalšie dve polia, až kým sa nakoniec vytvorí toľko polí koľko je prvkov vo vstupnom poli. Algoritmus využíva vedomosť, že jednoprvkové pole je vždy utriedené.

Ďalším dôležitým krokom je tieto jednoprvkové polia spojiť. Na to slúži špeciálna funkcia, ktorá môže mať tvar, kde sa porovnávajú prvé prvky z oboch polí a ten menší je pridaný do výsledného poľa, pričom je z pôvodného poľa vymazaný.

Časová zložitosť

V najhoršom prípade: O(n log n) – časová zložitosť Merge Sort je vo všetkých troch prípadoch rovnaký

V najlepšom prípade: Ω(n log n)

Priemerný prípad: Θ(n log n)

Pamäťová zložitosť

Ako najväčšia nevýhoda algoritmu Merge Sort je práve jeho pamäťová zložitosť, ktorá je spôsobená práve kvôli jeho princípu rekurzívneho volania funkcie.

Výhody a využitie

Veľkou výhodou oproti ostatným algoritmom je to, že čas, ktorý je potrebný pre usporiadanie je takmer nezávislý na začiatočnom zoradení postupnosti.

Merge Sort je základná metóda pre usporadúvanie sekvenčných súborov.

V mnohých programovacích jazykov je Merge Sort implicitným triediacim algoritmom (v Perl 5.8, v Jave alebo v GNU C Library).

Pseudo kód triediaceho algoritmu Merge Sort

```
funkcia merge_sort(IT = nezoradené pole prvkov)  {
        pole jedna, dva
        spravodlivo rozdeľ prvky podľa indexu do polí jedna a dva
        merge_sort(jedna)
        merge_sort(dva)
        pole výsledok
        výsledok = merge(jedna, dva)
}
funkcia merge(jedna = utriedené pole, dva = utriedené pole)  {
        pole výsledok  kým majú obe polia jedna a dva aspoň jeden prvok   {
                ak je väčší prvý prvok z poľa jedna než prvý prvok z poľa dva    {
                        pridaj do výsledku prvý prvok z jedna a vymaž ho z jedna
                }   inak   {
                        pridaj do výsledku prvý prvok z dva a vymaž ho z dva
                }
        }   ak sa stalo, že v jednom poli bolo o jeden viac prvkov než v druhom, pridaj tento
        prvok na koniec výsledku
        vráť výsledok
} [7]
```

Vizualizácia postupu triedenia algoritmom Merge Sort

Utriedenie nasledujúceho číselného poľa algoritmom Merge Sort.

Merge Sort najskôr rozdelí celé pole na dve rovnaké časti. V tomto prípade je 8 prvkové pole rozdelené na dve časti po 4 prvky.

Ako ďalšie sú rozdelené aj tieto polia na polovice.

A tak isto rozdelíme aj tieto menšie polovice, až pole rozdelíme na základné prvky.

Teraz budú prvky spájané v rovnakých pároch ako boli rozdelené. Na obrázku to je vidno pomocou farebného označenia. Najskôr algoritmus spája 14 a 33, pričom tieto čísla už sú v správnom poradí. Ďalej sa porovnáva 27 a 10 a tieto dva prvky spojíme do spoločného poľa, kde najskôr umiestnime 10 a potom 27. Potom sa zmení poradie čísiel 19 a 35 a taktiež sa spoja čísla 42 a 44.

V ďalšej iterácii sú porovnávané dvojprvkové polia a následne na to spájané do nového poľa kde sú tieto prvky správne utriedené.

Po poslednom spájaní vyzerá pole nasledovne -

3 Metodika práce a metódy skúmania

Metodika práce bola realizovaná vytvorením programu v programovacom jazyku C na porovnanie exekučnej efektívnosti Insertion Sort a Selection Sort usporadúvajúcich veľké súbory dát.

Testovanie bolo vykonané na notebooku ASUS X555LJ s nasledujúcimi parametrami:

- Procesor: Intel(R) Core(TM) i3-5010U CPU @ 2.10GHz 2.10GHz
- RAM: 8.00 GB
- Typ systému: 64-bitový operačný systém, Windows 10

Projekt bol vytvorený v prostredí textového editora Sublime Text 2, ktorý slúži na zvýraznenie syntaxe programovacích jazykov. Ku tomuto textovému editoru bol pripojený kompilátor GCC 7.3, pomocou ktorého bol celý projekt kompilovaný.

4 Cieľ práce

Bakalárska práca je rozdelená do viacerých častí, pričom praktická časť bakalárskej práce sa zameriava na opis vytvoreného programu v jazyku C, ktorý porovnáva priemerný exekučný čas triediacich algoritmov Insertion Sort a Shell Sort, ktoré triedia dynamicky vytvorené sto tisíc prvkové pole, ktoré je naplnené pseudo-náhodnými číslami. Na základe tohto merania vieme porovnať efektívnosť týchto dvoch triediacich algoritmov.

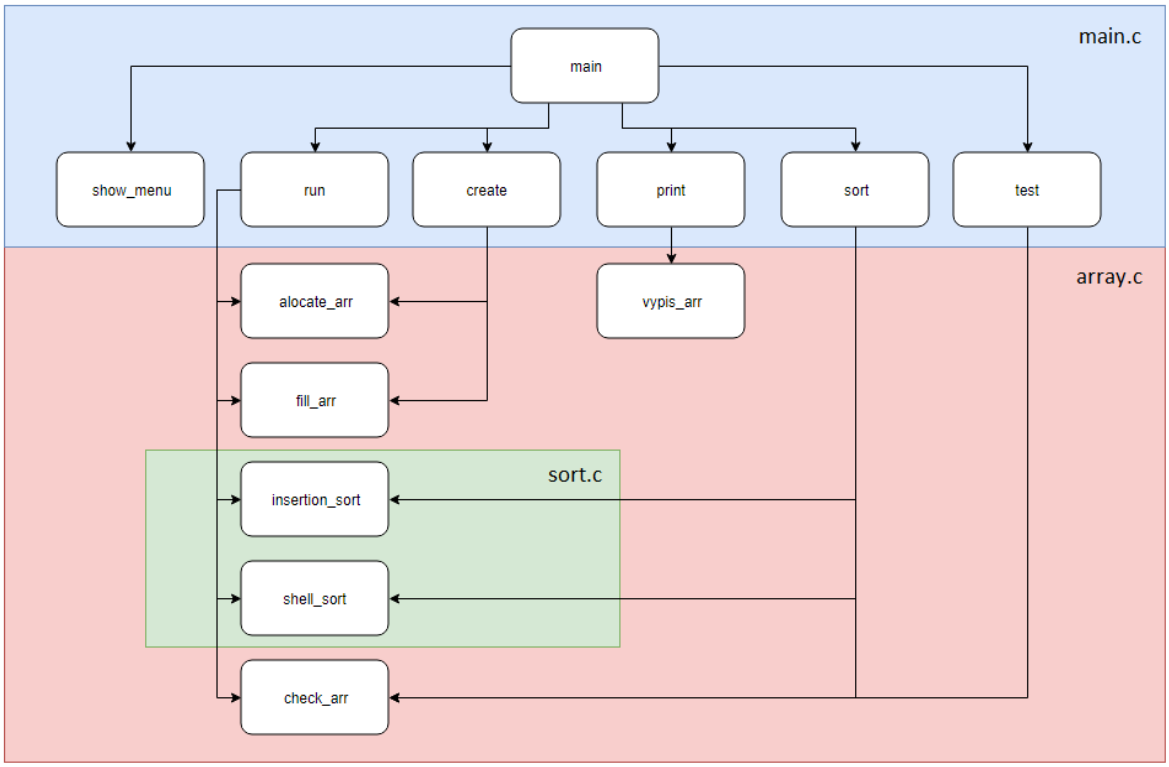
Ďalšie časti bakalárskej práce budú zamerané na metodiku práce, metódy skúmania a výsledky výskumu.

5 Popis programu

Program na porovnávanie exekučnej efektívnosti triediacich algoritmov Insertion Sort a Shell Sort pozostáva z 3 častí. Tieto 3 časti sú rozdelené do troch zdrojových súborov tak, aby bola zachovaná prehľadnosť kódu. Na nasledujúcom obrázku je zobrazená mapa volaných funkcií a zároveň je na obrázku zaznačené, v akom zdrojovom súbore sa jednotlivé funkcie nachádzajú. Ako je z obrázka možné vyčítať, funkcie v zdrojovom súbore *main.c*

slúžia na volanie ostatných funkcií zo zdrojových súborov *array.c* a *sort.c*. V zdrojovom súbore *array.c* sa nachádzajú funkcie slúžiace na vytvorenie, naplnenie a prácu s polom. V zdrojovom súbore *sort.c* sa nachádzajú iba dve funkcie a to konkrétne funkcie triediacich algoritmov Insertion Sort a Shell Sort.

Takto navrhnutý program je v budúcnosti ľahké modifikovať, respektíve rozšíriť o iné triediace algoritmy.



Obrázok č. 1: Mapa volaní funkcií v programe [Zdroj: Vlastné spracovanie]

Prvá časť je v zdrojovom súbore *main.c*, kde sa nachádza celkovo 7 funkcií, ktoré zabezpečujú celkový chod programu a volanie ostatných funkcií.

Na začiatku si do projektu vložíme všetky knižnice, ktoré bude projekt potrebovať, a to pomocou príkazov *#include <stdio.h>*, ktorá slúži ako základná knižnica jazyka C pre vstup a výstup, *#include <time.h>*, ktorý je v projekte použitá na meranie času a taktiež *#include <stdlib.h>*, pomocou ktorého sa budú generované náhodné čísla.

Prepojenie zdrojových súborov funguje pomocou použitia príkazov *#include "array.c"* a *#include "sort.c"* na začiatku hlavného zdrojového súboru *main.c*.

Ďalej nasledujú ostatné funkcie:

void show_menu()
*void run(int ** arr1, int ** arr2, int n, int x, int count)*
*void create(int ** arr1, int ** arr2, int n, int x)*
*void print(int * arr1, int * arr2, int n)*
*void sort(int ** arr1, int ** arr2, int n)*
*void test(int ** arr1, int ** arr2, int n)*
int main()

Funkcia *void show_menu()* slúži na zobrazenie menu, pomocou ktorého sa spúšťajú jednotlivé voľby programu.

```
void  show_menu()  {
   printf(">>>>>>>>>>MENU<<<<<<<<<<\n");
   printf("r - spustit program sekvencne\n");          //run
   printf("c - vytvorit pole\n");                      //create
   printf("p - vypisat pole\n");                       //print
   printf("s - zoradit pole\n");                       //sort
   printf("t - kontrola pola\n");                      //test
   printf("m - menu\n");                               //menu
   printf("e - koniec\n");                             //end
}
```

Obrázok č. 2: Zdrojový kód funkcie *show_menu* [Zdroj: Vlastné spracovanie]

Funkcia má návratovú hodnotu typu void, pretože využíva len funkciu printf() na vypísanie jednotlivých volieb, ktoré sú označene kódom písmena.

Funkcia *void run(int **arr1, int **arr2, int n, int x, int count)*, spúšťaná klávesou *r*, slúži na spustenie programu v automatickom režime, kedy po zadaní veľkosti generovaného poľa - *n*, hornej hranice náhodne generovaných čísel - *x* a počtu opakovaní - *count*, automaticky spustí triedenie polí.

Na začiatku sú deklarovaná premenná *int test_result*, ktorá bude slúžiť ako pomocná premenná do ktorej si budeme ukladať návratovú hodnotu funkcie *check_arr(*arr, n)*, ktorá slúži na overenie správnosti usporiadania poľa. Ďalej sú definované dve premenné *float avg_time1, float avg_time2*, ktoré program využíva na ukladanie priemerného času, ktorý bol potrebný na usporiadanie poľa. Premenné *time_t current_time* a ukazovateľ *char* c_time_string* slúžia na meranie času. Do *current_time sa* najskôr uloží presný čas, ktorý je

potom ako argument poslaný do funcie *ctime()*, ktorá vracia zadaný čas v podobe reťazca. Taktiež je v úvode tejto funkcie deklarovaný ukazovateľ *FILE *file_vysledky*, ktorý bude slúžiť na prácu s výstupným súborom, do ktorého sa ukladajú výsledky. Tento súbor je otvorený s parametrom *a* – pripisovanie na koniec súboru.

Najskôr je potrebné alokovať pamäť pre obe polia **arr1*, **arr2* pomocou funkcie *allocate_arr(*arr, n)*. Takto vytvorené pole je ďalej naplnené náhodnými číslami v intervale (0, *x*) pomocou funkcie *fill_arr(*arr1,*arr2,n,x)*. Táto funkcia naplní obe polia rovnakými číslami, aby bolo možné presné porovnanie exekučných časov identických polí. Potom nasleduje samotné usporiadanie prvého poľa funkciou *shell_sort(*arr1, n)* a druhého poľa funkciou *insertion_sort(*arr2,n)*. V oboch prípadoch je meraný čas, za ktorý sa danej funkcii podarilo usporiadať pole, a taktiež je pre obe polia skontrolovaná správnosť usporiadania. V prípade ak sa pole nepodarilo usporiadať, do konzoly je vypísaná chybová správa s uvedeným indexom prvého nesprávne zoradeného prvku.

Táto časť kódu sa opakuje *count*-krát pomocou cyklu *for*.

Na konci sú výsledky o priemernom trvaní funkcií na utriedenie polí vypísané do konzoly a taktiež sú zapísané do výstupného súboru.

```
void run(int ** arr1, int ** arr2, int n, int x, int count) {
    int test_result;
    float avg_time1 = 0.0, avg_time2 = 0.0;
    time_t current_time;
    char * c_time_string;
    FILE * file_vysledky;

    current_time = time(NULL);
    c_time_string = ctime( & current_time);
    file_vysledky = fopen("history.log", "a");

   * arr1 = allocate_arr( * arr1, n);
   * arr2 = allocate_arr( * arr2, n);
    for (int i = 0; i < count; ++i) {
      fill_arr( * arr1, * arr2, n, x);
      clock_t time;
          /*TEST SORTING*/
          /*Shell sort*/
      time = clock();
      shell_sort( * arr1, n);
      time = clock() - time;
      test_result = check_arr( * arr1, n);
      if (test_result == -1) {
        printf("Pole[%d] je zoradene (0->9)\n", i);
```

```
        } else {
            printf("Pole[%d] NIE JE zoradene\n", i);
        }
        avg_time1 = avg_time1 + time;
        printf("SHELLSORT trval %f \n",  ((float) time) / CLOCKS_PER_SEC);
        /*Insertion sort*/
        time = clock();
        insertion_sort( * arr2, n);
        time = clock() - time;
        test_result = check_arr( * arr2, n);
        if (test_result == -1) {
            printf("Pole[%d] je zoradene (0->9)\n", i);
        } else {
            printf("Pole[%d] NIE JE zoradene\n", i);
        }
        avg_time2 = avg_time2 + time;
        printf("INSERTIONSORT trval %f \n\n", ((float) time) / CLOCKS_PER_SEC);
    }

    /* PRINT OUT */
    printf("_____\n");
    printf("---------SHELLSORT---------\n");
    printf("Priemerny cas: %f\n", (((float) avg_time1) / CLOCKS_PER_SEC) / count);
    printf("_____\n");
    printf("-------INSERTIONSORT-------\n");
    printf("Priemerny cas: %f\n", (((float) avg_time2) / CLOCKS_PER_SEC) / count);

    /*PRINT INTO FILE*/
    fprintf(file_vysledky, "Date: \t\t\t %s", c_time_string);
    fprintf(file_vysledky, "n: \t\t\t\t %d\n", n);
    fprintf(file_vysledky, "x: \t\t\t\t %d\n", x);
    fprintf(file_vysledky, "count: \t\t\t %d\n", count);
    fprintf(file_vysledky, "shell_sort: \t %f\n", (((float) avg_time1) /
    CLOCKS_PER_SEC) / count);
    fprintf(file_vysledky, "insertion_sort:  %f\n", (((float) avg_time2) /
    CLOCKS_PER_SEC) / count);
    fprintf(file_vysledky, "-----------------\n");
    fclose(file_vysledky);
}
```

Obrázok č. 3: Zdrojový kód funkcie *run* [Zdroj: Vlastné spracovanie]

Ďalšie funkcie slúžia na manuálne spustenie programu, kedy sa vytvorí iba jedna dvojica dynamicky alokovaného poľa podľa užívateľa zadaných parametrov, ďalšou voľbou sa vopred alokované pole naplní pseudo-náhodnými číslami a ďalšou voľbou sa toto pole utriedia pomocou algoritmov Insertion Sort a Shell Sort. Kontrolu usporiadania týchto polí je možné spustiť osobitnou voľbou, tak isto, ako je možné dané polia vypísať do konzoly.

Funkcia *void create(int **arr1, int **arr2, int n, int x)* slúži na vytvorenie dvoch polí *arr1* a *arr2*, kedy sa dané polia najskôr alokujú a potom naplnia na základe parametrov *n, x*. Obe polia sú alokované dynamicky, ale nezávisle na seba, čiže oba smerníky ukazujú na rozdielne polia.

```
void create(int * * arr1, int * * arr2, int n, int x) {
        * arr1 = allocate_arr( * arr1, n);
        * arr2 = allocate_arr( * arr2, n);
        fill_arr( * arr1, * arr2, n, x);
 }
```

Obrázok č. 4: Zdrojový kód funkcie *create* [Zdroj: Vlastné spracovanie]

Funkcia *void sort(int **arr1, int **arr2, int n)* slúži na zavolanie funkcií, ktoré polia utriedia pomocou algoritmov Insertion Sort a Shell Sort a zmerajú ich exekučný čas. Ešte pred spustením tejto funkcie z menu je vždy skontrolované, či boli predtým polia skutočne vytvorené, aby sa zabránilo neoprávnenému pristupovaniu k pamäti.

```
void sort(int * * arr1, int * * arr2, int n) {
    clock_t time;
    time = clock();
    shell_sort( * arr1, n);
    time = clock() - time;
    if (check_arr( * arr1, n)) {
       printf("Pole[1] je zoradene (0->9)\n");
    } else {
       printf("Pole[1] NIE JE zoradene\n");
    } //avg_time1 = avg_time1 + time;
   printf("SHELLSORT trval %f \n",  ((float) time) / CLOCKS_PER_SEC);
    time = clock();
    insertion_sort( * arr2, n);
    time = clock() - time;
    if (check_arr( * arr2, n)) {
       printf("Pole[2] je zoradene (0->9)\n");
    } else {
       printf("Pole[2] NIE JE zoradene\n");
```

```
    } //avg_time2 = avg_time2 + time;
    printf("INSERTIONSORT trval %f \n\n", ((float) time) / CLOCKS_PER_SEC);
}
```

Obrázok č. 5: Zdrojový kód funkcie *sort* [Zdroj: Vlastné spracovanie]

Funkcia *void print(int *arr1, int *arr2, int n)* slúži na výpis polí *arr1* a *arr2* v podobe reťazca pomocou funkcie *vypis_arr(arr,n)*, ktorá sa nachádza v zdrojovom súbore *array.c*.

```
void print(int * arr1, int * arr2, int n) {
    printf("ARR1:\n");
    vypis_arr(arr1, n);
    printf("\n\nARR2:\n");
    vypis_arr(arr2, n);
    printf("\n");
}
```

Obrázok č. 6: Zdrojový kód funkcie *print* [Zdroj: Vlastné spracovanie]

Funkcia *void test(int **arr1, int **arr2, int n)* slúži na kontrolu toho, či sú polia *arr1*, *arr2* utriedené pomocou funkcie *check_arr(*arr1, n)* a v prípade, že to tak nie je, program vypíše do konzoly chybovú hlášku s pozíciou prvého nesprávne usporiadaného prvku.

```
void test(int * * arr1, int * * arr2, int n) {
    int test_result;
    test_result = check_arr( * arr1, n);
    if (test_result == -1) {
        printf("Pole[1] je zoradene (0->9)\n");
    } else if (test_result >= 0) {
        printf("Pole[1] NIE JE zoradene\nPrvy chybny prvok na indexe:  %d\n", test_result);
    }
    test_result = check_arr( * arr2, n);
    if (test_result == -1) {
          printf("Pole[2] je zoradene (0->9)\n");
    } else if (test_result >= 0) {
        printf("Pole[2] NIE JE zoradene\nPrvy chybny prvok na indexe:  %d\n", test_result);
   }
}
```

Obrázok č. 7: Zdrojový kód funkcie *test* [Zdroj: Vlastné spracovanie]

Ako posledná sa v zdrojovom súbore *main.c* nachádza funckia *int main()*, ktorá je hlavná funkcia celého programu a spúšťa sa ako prvá. Najskôr sú definované dva smerníky *int *p_arr1, int *p_arr2*, v ktorých budú uložené adresy polí, s ktorými ostatné funkcie pracujú.

Ďalej je zobrazené menu pomocou funkcie *show_menu()* a potom nasleduje nekonečný cyklus, v ktorom môže užívateľ vyberať voľbu, ktorá sa má pustiť. Taktiež sa v tejto funkcii požaduje od užívateľa zadať počet prvkov v poli - *n*, horná hranica náhodne generovaných čísiel - *x*, prípadne počet opakovaní - *count*. Tento cyklus je možné opustiť po vybratí voľby *e*. Následne sa uvoľní pamäť, ktorá bola potrebná pre polia.

Druhá časť programu je v súbore *array.c*, kde sa nachádzajú všetky funkcie na prácu s poľom. Obsahuje celkovo 4 funkcie:

*int * allocate_arr(int * arr, int n)*
*void fill_arr(int * arr1, int * arr2, int n, int scope)*
*int check_arr(int * arr, int n)*
*void vypis_arr(int * arr, int n)*
Prvá funkcia *int * allocate_arr(int *arr, int n)* slúži na alokovanie pamäte poľa pomocou funkcie malloc(). Funkcia vracia smerník na takto alokované miesto v pamäti.

```
int * allocate_arr(int * arr, int n) {
  arr = (int * ) malloc(n * sizeof(int));
  return arr;
}
```

Obrázok č. 8: Zdrojový kód funkcie *allocate_arr* [Zdroj: Vlastné spracovanie]

Funkcia *void fill_arr(int *arr1, int *arr2, int n, int scope)* slúži na naplnenie poľa náhodnými číslami podľa intervalu.

```
void fill_arr(int * arr1, int * arr2, int n, int scope) {
  srand(time(NULL));
  for (int i = 0; i < n; i++) {
    arr1[i] = rand() % scope + 1;
    arr2[i] = arr1[i];
  }
}
```

Obrázok č. 9: Zdrojový kód funkcie *fill_arr* [Zdroj: Vlastné spracovanie]

Funkcia *int check_arr(int *arr, int n)* slúži kontrolu toho, či je pole usporiadané zostupne. Funkcia porovnáva dve susediace hodnoty a v prípade, že je druhý prvok menší ako prvý, pole teda nie je usporiadané správne, funkcia vracia číslo indexu druhého prvku. To slúži v programe na to, aby bolo možné ľahko identifikovať miesto, kde pole nie je usporiadané. V prípade, že je pole usporiadané správne, čiže všetky prvky sú usporiadané zostupne, funkcia vracia hodnotu -1.

```
int check_arr(int * arr, int n) {
   for (int i = 0; i < n - 1; ++i) {
      if ( * (arr + i) > * (arr + i + 1)) return i + 1;
   }
   return -1;
 }
```

Obrázok č. 10: Zdrojový kód funkcie *check_arr* [Zdroj: Vlastné spracovanie]

Tu ešte treba pripomenúť, že podľa programátorskej praxe sa pri funkcii, ktorá overuje podmienku, pracuje najčastejšie s dátovým typom *boolean*, ktorý je funkciou vrátený – pravda (1), ak je podmienka splnená a nepravda (0), ak podmienka nie je splnená. Tento typ však programovací jazyk C nepodporuje a preto bol použitý dátový typ *int.*

Funkcia *void vypis_arr(int *arr, int n)* slúži na vypísanie hodnôt poľa do konzoly pomocou cyklu *for*, pričom v každej iterácii je vypísaná hodnota daného indexu poľa.

```
void vypis_arr(int * arr, int n) {
   for (int i = 0; i < n; ++i) {
      printf("%d ", arr[i]);
   }
 }
```

Obrázok č. 11: Zdrojový kód funkcie *vypis_arr* [Zdroj: Vlastné spracovanie]

Posledná časť programu sa nachádza v zdrojovom súbore *sort.c*. Nachádzajú sa tu obe funkcie, ktoré program používa na triedenie. Výhodou tohto oddelenia triediacich algoritmov do osobitného zdrojového súboru je v tom, že je zachovaná prehľadnosť kódu a taktiež výhodné rozdelenie programu do logických častí.

*void shell_sort(int * arr, int n)*
*void insertion_sort(int * arr, int n)*

V úvode funkcie *void shell_sort(int *arr, int n)* sú deklarované 3 premenné typu *int – i, j, k, tmp.* Premenná *i* slúži ako pomocná premenná, pomocou ktorej iterujeme prvý cyklus *for* a zároveň predstavuje medzeru medzi dvoma prvkami, ktoré budú porovnávané, a preto musí spĺňať podmienku, že medzera musí byť väčšia ako 0. Po každej iterácii sa premenná *i* zmenší o polovicu.

Ďalej nasleduje vnorený cyklus *for,* ktorý používa na iteráciu premennú *j,* do ktorej je v inicializačnej časti cyklu zapísaná hodnota premennej *i.* Pomocou tohto cyklu algoritmus prechádza celé pole. Potom nasleduje cyklus, v ktorom sa kontrolujú všetky prvky, medzi ktorými je daná medzera a tieto prvky sú porovnané. Ak sa tieto prvky nachádzajú v nesprávnom poradí sú navzájom vymenené pomocou pomocnej premennej *tmp*.

```
void shell_sort(int *arr, int n){
        int i, j, k, tmp;
        for (i = n / 2; i > 0; i = i / 2) {
                for (j = i; j < n; j++){
                        for(k = j - i; k >= 0; k = k - i){
                                if (arr[k+i] >= arr[k])
                                        break;
                                else
                                {
                                        tmp = arr[k];
                                        arr[k] = arr[k+i];
                                        arr[k+i] = tmp;
                                }
                        }
                }
        }
}
```

Obrázok č. 12: Zdrojový kód funkcie *shell_sort* [Zdroj: Vlastné spracovanie]

Toto riešenie je náročnejšie ako Insertion sort, avšak odstraňuje problém, kedy sú porovnávané iba prvky, ktoré spolu susedia. Takto optimalizované riešenie Shell Sort je preto oproti Insertion Sort vhodnejšie na triedenie veľkého množstva prkov.

Vo funkcii *void insertion_sort(int *arr, int n)* sú najskôr deklarované 3 premenné – *i, key, j*. Funkcia ďalej vchádza do cyklu *for*, ktorý bude iterovaný pomocou premennej *i*, do ktorej je najskôr v inicializačnej časti uložená hodnota 1. V podmienkovej časti je určené, že premenná *i* nemôže byť väčšia ako premenná *n*, a tým je zaručené, že všetky iterácie sa budú týkať iba prvkov v poli *arr*. V inkrementačnej časti je určené, že po každej iterácii je hodnota premennej *i* zväčšená o 1. Ďalej je do premennej *key* uložená hodnota prvku poľa *arr* na pozícii *i*. Do pomocnej premennej *j* následne uložené číslo, ktoré reprezentuje o jeden menší index, aký má v danej iterácii prvok, ktorý sa uložil do premennej *key*, a teda horný index utriedenej časti poľa. Pre takto uložený prvok v pomocnej premennej *key*, hľadá algoritmus pomocou vnoreného cyklu *while* pozíciu, kde patrí. Pomocou cyklu *while* sa taktiež všetky prvky, ktoré sú väčšie ako hodnota premennej *key* posúvajú v poli o jednu pozíciu doprava. Keď algoritmus narazí na prvok, ktorý je menší ako *key*, uloží *key* za tento prvok a tým sa skončí jedna iterácia prvého cyklu *for*.

```
void insertion_sort(int *arr, int n){
        int i, key, j;
        for (i = 1; i < n; i++){
                key = arr[i];
                j = i-1;
                while (j >= 0 && arr[j] > key){
                        arr[j+1] = arr[j];
                        j = j-1;
                }
                arr[j+1] = key;
        }
}
```

Obrázok č. 13: Zdrojový kód funkcie *insertion_sort* [Zdroj: Vlastné spracovanie]

Toto riešenie je jednoduché a ľahko zrozumiteľné, avšak treba si hneď uvedomiť, že pre pole, ktoré by obsahovalo veľké množstvo prvkov, by bolo potrebné na nájdenie správnej pozície pre malý prvok, ktorý by bol umiestnený na konci poľa, urobiť toľko porovnávaní, koľko je väčších prvkov pred ním. Preto sa tento algoritmus neodporúča používať pri veľkých množstvách prvkov.

Obe funkcie využívajú na vymenenie prvkov v poli iba jednu pomocnú premennú, čo potvrdzuje, že oba algoritmy majú konštantnú pamäťovú náročnosť. Táto vlastnosť sa považuje za najväčšiu výhodu algoritmov Shell Sort a Insertion Sort.

6 Výsledky práce

Ako už bolo opísané v predošlých kapitolách, v programe je meraný čas potrebný na usporiadanie poľa prvkov, či už pomocou triediaceho algoritmu Insertion Sort alebo pomocou Shell Sort algoritmu a v prípade, že je program spustený v režime, kedy sa tieto triedenia opakujú, je zaznamenávaný aj priemerný čas.

Tieto časy sú buď vypísané do konzoly, respektíve sú zapísané do výstupného súboru tak, aby bola používateľovi poskytnutá história spustení triedenia.

```
Výsledky z konzoly:
Pocet prvkov v poli:
100000
Zadajte hornu hranicu generovanych cisel:
1000
Zadajte pocet opakovani:
50
Pole[0] je zoradene (0->9)
SHELLSORT trval 0.037000
Pole[0] je zoradene (0->9)
INSERTIONSORT trval 10.985000

Pole[1] je zoradene (0->9)
SHELLSORT trval 0.047000
Pole[1] je zoradene (0->9)
INSERTIONSORT trval 10.834000
...
Pole[49] je zoradene (0->9)
SHELLSORT trval 0.039000
Pole[49] je zoradene (0->9)
INSERTIONSORT trval 11.115000
```

```
_____________________________
---------SHELLSORT---------
Priemerny cas: 0.040860
_____________________________
-------INSERTIONSORT-------
Priemerny cas: 11.234260
R – DONE
```

Obrázok č. 14: Výstup programu do konzoly [Zdroj: Vlastné spracovanie]

Vo výstupe je možné vidieť základné informácie o priebehu programu, kedy je od užívateľa najskôr vyžiadané zadať základné parametre – počet prvkov v poli, horná hranica generovaných čísiel a počet opakovaní. Ďalej sa automaticky spustilo triedenie pomocou triediacich algoritmov Shell Sort a Insertion Sort, pričom bol výsledok každého jedného triedenie vypísaný do konzoly. Z výpisu je možné zistiť, ktoré pole je v tom momente triedené a po utriedení je do konzoly vypísaný, či bolo triedenie úspešné a aký čas bol potrebný na toto utriedenie.

Toto spustenie programu sa taktiež uložilo aj do výstupného súboru, kde sú zapísané základné parametre, s ktorými bol program spustený a taktiež priemerný čas potrebný na utriedenie polí. Taktiež je výsledok každého testu označený dátumom a časom spustenia.

```
-----------------------------------------------------
Date:               Tue Apr 17 14:18:20 2018
n:                  100000
x:                  1000
count:              50
shell_sort:         0.040860
insertion_sort:     11.234260
-----------------------------------------------------
```

Obrázok č. 15: Výstup programu do textového súboru [Zdroj: Vlastné spracovanie]

Z výsledkov je vidieť, že priemerný čas utriedenia poľa, ktorý obsahuje 100 000 prvkov náhodne generovaných v rozsahu 0 – 1 000, trvá algoritmu Shell Sort 0.04086 sekúnd a algoritmu Insertion Sort trvá zoradenie 11,23426 sekúnd.

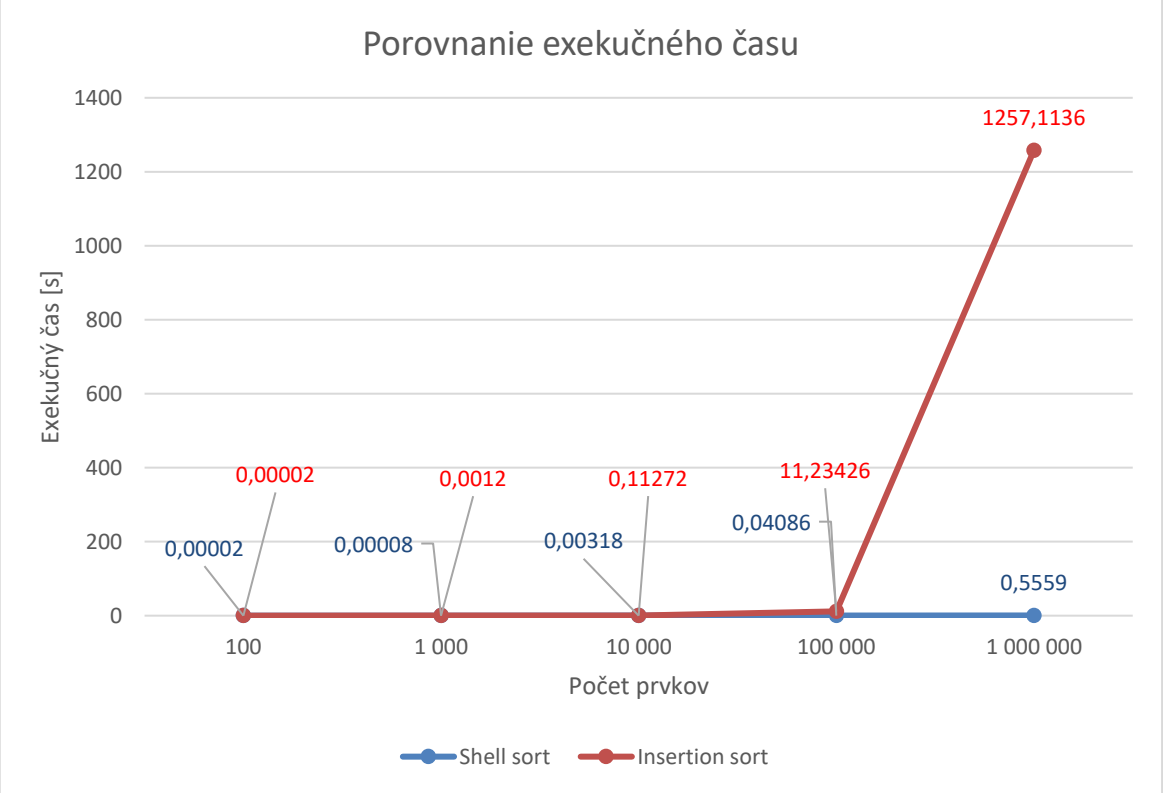
Ďalej sme robili rôzne testovania pri rôznom počte prvkov, pričom rozsah náhodne generovaných čísiel ostal konštantný – od 0 do 1000.

Z výsledkov bola vytvorená nasledujúca tabuľka a ku každému testu porovnávací stĺpcový graf.

Počet prvkov	Shell Sort [s]	Insertion Sort [s]
100	0,00002	0,00002
1 000	0,00008	0,0012
10 000	0,00318	0,11272
100 000	0,04086	11,23426
1 000 000	0,5559	1257,1136

Tabuľka č. 1: Porovnanie priemerných časov [Zdroj: Vlastné spracovanie]

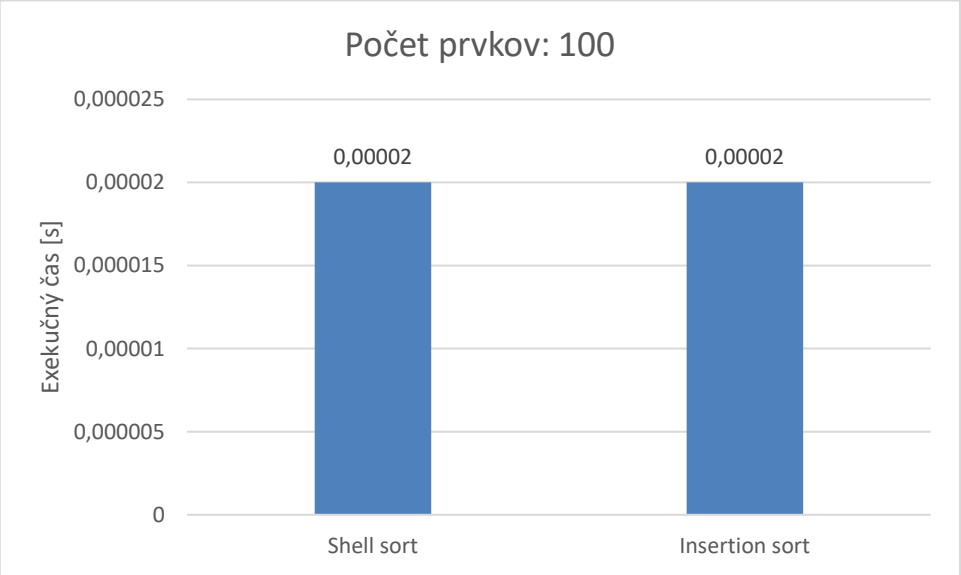
Z tabuľky je vidno, že výsledky medzi Shell Sort a Insertion Sort sú veľmi rozdieľne. Z týchto údajov bol vytvorený nasledujúci graf.



Graf č. 1: Porovnanie exekučného času [Zdroj: Vlastné spracovanie]

Z grafu hneď na prvý pohľad vyplýva, že ako sa počet prvkov v triedenom poli zväčšuje, tak sa aj zvyšuje časová náročnosť triedenia. V prípade Insersion Sort je tento rozdiel dokonca taký výrazný, že pri väčšom počte prvkov dosahuje exekučný čas niekoľko sekúnd.

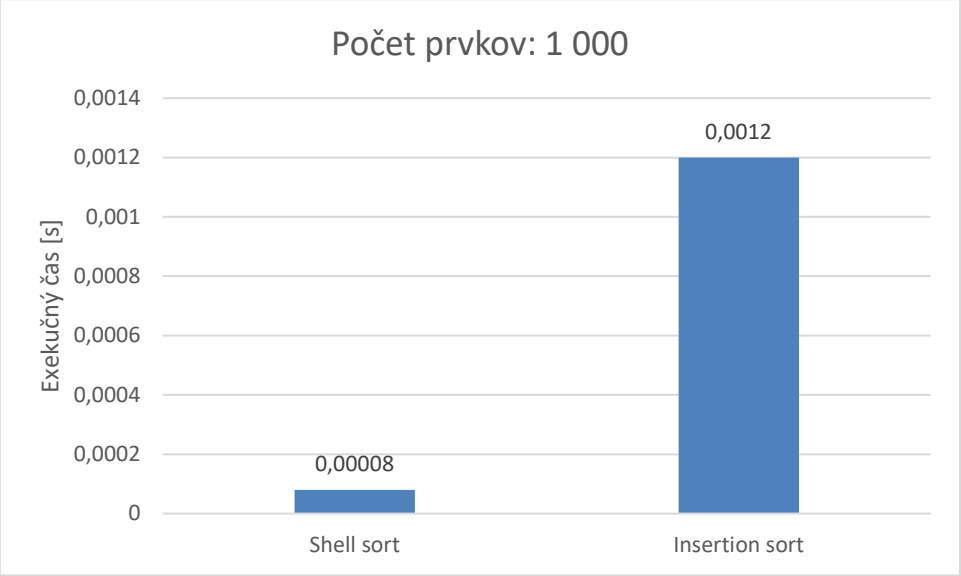
V prípade, kedy bolo testovaných počet prvkov 100 bol priemerný čas algoritmu Shell Sort a Insertion Sort rovnaký, a to 0,00002 sekundy.



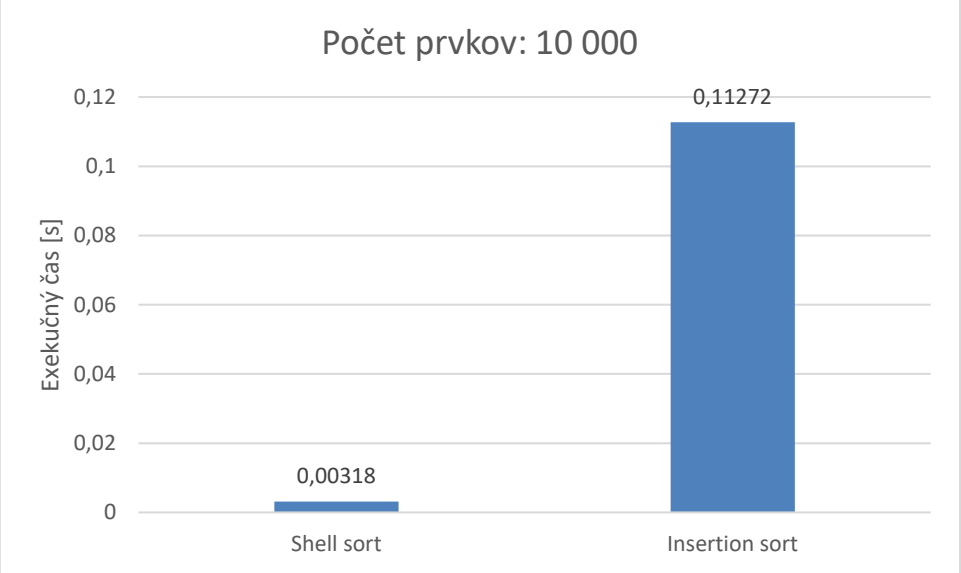
Graf č. 2: Porovnanie priemerného času pri 100 prvkoch [Zdroj: Vlastné spracovanie]

Pri nízkom počte prvkov, ktoré treba usporiadať pomocou algoritmov Insertion Sort a Shell Sort sú oba algoritmy efektívne, oba algoritmy vykonajú približne rovnaký počet porovnaní a výmen. Preto je exekučný čas pre obidva algoritmy rovnaký.

Pri počte prvkov 1 000 bol test stále veľmi rýchly, rozdiel medzi algoritmami Insertion Sort a Shell Sort bol len 0,00112, čo ale predstavuje, že v tomto teste je algoritmus Insertion Sort 15-krát rýchlejší ako algoritmus Shell Sort. Priemerný čas algoritmu Insertion Sort bol 0,0012 sekundy a pre algoritmus Shell Sort 0,00008 sekundy.

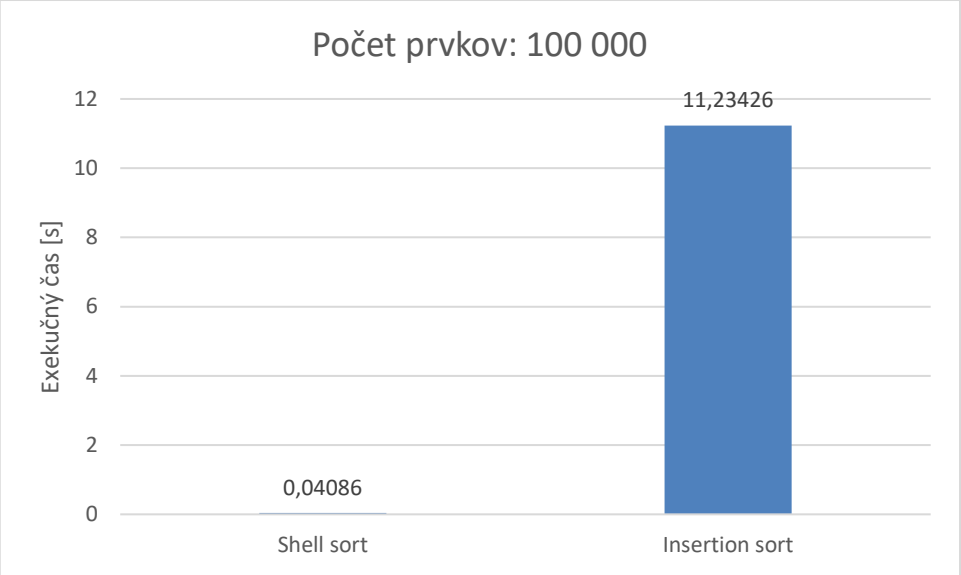


Graf č. 3: Porovnanie priemerného času pri 1 000 prvkoch [Zdroj: Vlastné spracovanie]



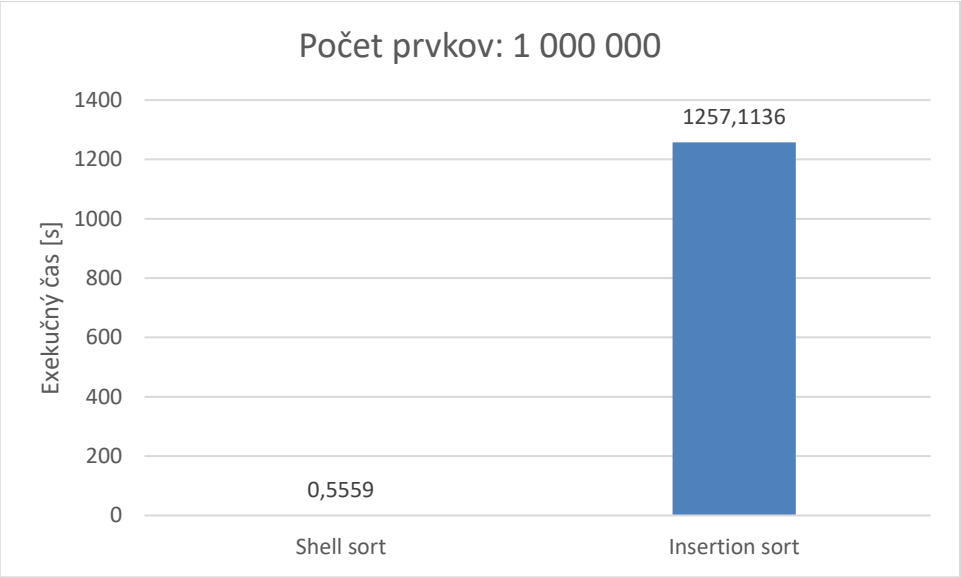
Graf č. 4: Porovnanie priemerného času pri 10 000 prvkoch [Zdroj: Vlastné spracovanie]

Pri počte 10 000 prvkov trval test Shell Sort oproti testu s 1 000 prvkami 40-krát dlhšie, avšak pre Insertion Sort to bolo až 94-krát dlhšie. Priemerný čas algoritmu Insertion Sort bol 0,11272 sekundy a pre algoritmus Shell Sort 0,00318 sekundy.



Graf č. 5: Porovnanie priemerného času pri 100 000 prvkoch [Zdroj: Vlastné spracovanie]

Pri počte 100 000 prvkov je rozdiel ešte väčší. Priemerný čas algoritmu Shell Sort trval 0,04086 sekundy a Insertion Sort trval 11,23426 sekúnd. To znamená, že na usporiadanie jedného prvku pri použití algoritmu Shell Sort bolo potrebných približne 0,0000004 sekundy a pri použití Insertion Sort až 0,0001 sekundy. Z uvedených výsledkov teda vyplýva, že na utriedenie poľa s počtom prvkov viac ako 10 000 nie je vhodné použiť triediaci algoritmus Insersion Sort. Shell Sort práve vďaka svojmu vylepšeniu oproti Insertion Sort zvláda takéto veľké pole utriediť vo výrazne menšom exekučnom čase a preto je na toto triedenie vhodnejšie.



Graf č. 6: Porovnanie priemerného času pri 1 000 000 prvkoch [Zdroj: Vlastné spracovanie]

Ako posledné bolo testované usporadúvanie poľa, ktorý obsahoval 1 milión náhodne vygenerovaných čísiel, pričom priemerný čas potrebný na usporiadanie tohto poľa trvalo algoritmu Shell Sort 0,5559 sekundy a algoritmu Insertion Sort 1257,11 sekúnd, čo je viac ako 20 minút. Shell Sort je teda v tomto prípade až 2 260-krát rýchlejší ako Insertion Sort.

Keby sme v testoch pokračovali a zvyšovali by sme počet prvkov v poli na 10 000 000, respektíve 100 000 000, exekučný výpočtový čas Insertion Sort by začal byť nemerateľný a veľmi nežiadúci. Aj v prípade Shell Sort by bol čas potrebný na utriedenie prvkov v poli niekoľkonásobne dlhší, avšak oproti Insertion Sort by bolo zväčšenie zanedbateľné.

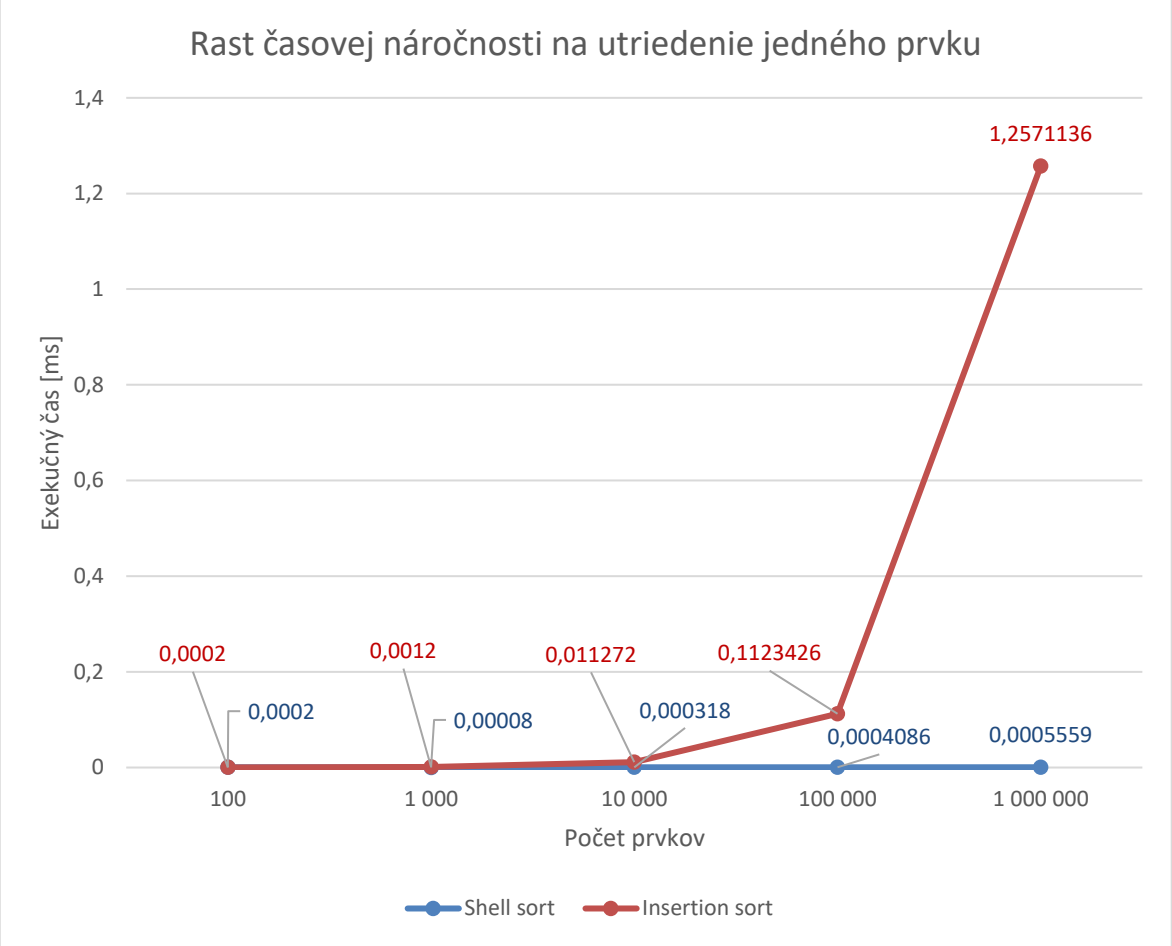
Taktiež môžeme pozorovať zvyšovanie časovej náročnosti na utriedenie jedného prvku pri zvyšujúcej sa veľkosti triedeného poľa. Časový rozdiel nenastal iba v prípade, keď bolo triedené pole so 100 prvkami, kedy obom algoritmom trvalo utriedenie jedného prvku rovnako a to 0,0002 ms.

	Čas potrebný na utriedenie jedného prvku [ms]	
Počet prvkov	Shell Sort	Insertion Sort
100	0,0002	0,0002
1 000	0,00008	0,0012
10 000	0,000318	0,011272
100 000	0,0004086	0,1123426
1 000 000	0,0005559	1,2571136

Tabuľka č. 2: Čas potrebný na utriedenie jedného prvku [Zdroj: Vlastné spracovanie]

Pri porovnávaní triediacich algoritmov Shell Sort a Insertion Sort sa pri zvyšovaní počtu prvkov zväčšuje aj čas potrebný na utriedenie jedného prvku. V prípade 100 000 prvkového poľa je Shell Sort rýchlejší približne 275-krát. Na všetkých týchto príkladoch môžeme vidieť ako je Shell Sort vhodnejší a efektívnejší na triedenie väčších súborov dát ako triediaci algoritmus Insertion Sort.

Z tabuľky č. 2 bol vytvorený graf, ktorý graficky znázorňuje rast časovej náročnosti potrebnej na utriedenie jedného prvku pri zvyšovaní počtu prvkov v poli.



Graf č. 7: Rast časovej náročnosti na utriedenie jedného prvku [Zdroj: Vlastné spracovanie]

Z týchto jednotlivých výsledkov testovania triediacich algoritmov je zrejmé, že pri väčšom počte prvkov v poli je Shell Sort časovo efektívnejší ako Insertion Sort a čím je počet prvkov poľa väčší, tak tým je väčší rozdiel medzi efektívnosťou jednotlivých algoritmov.

Záver

Cieľom tejto práce bol všeobecný prehľad triediacich algoritmov, ktoré pracujú na rôznom princípe. Pri algoritmoch bola opísaná ich časová zložitosť, pamäťová zložitosť a tak isto boli spomenuté ich najväčšie výhody. Práca bola bližšie zameraná na triediace algoritmy Insertion Sort a Shell Sort a porovnanie ich exekučnej efektívnosti pri usporadúvaní veľkých súborov dát v programe napísanom v programovacom jazyku C. Algoritmy boli porovnávané najmä na základe priemerného exekučného času, potrebného na usporiadanie dynamicky alokovaného poľa, ktorý bol naplnený náhodne generovanými číslami. Z jednotlivých výsledkov testovania triediacich algoritmov je zrejmé, že pri väčšom počte prvkov v poli je Shell Sort efektívnejší ako Insertion Sort. Porovnania boli robené aj pri rôznych počtoch prvkov poľa a takýmito meraniami bolo zistené, že čím väčšie je pole, tým väčší je rozdiel v exekučnej efektívnosti porovnávaných triediacich algoritmov.

Zoznam použitej literatúry

[1] KNUTH, D. E. 1998. The Art of Computer Programming - volume 3, Sorting and Searching. Reading, Massachusetts : Addison-Wesley, 1998. ISBN 0-201-89685-0, 1

[2] Friend, E. Sorting on electronic computer systems. J. ACM 3 (1956), 134–168

[3] https://sk.wikipedia.org/wiki/Triediaci_algoritmus

[4] http://www2.fiit.stuba.sk/~pospichal/molnar/system/Algoritmy.html

[5] SEDGEWICK, R. a WAYNE, K. 2011. Algorithms. Fourth Edition. Upper Saddle River, NJ : Addison-Wesley, 2011. ISBN 0-321-57351-X, 243

[6] http://www2.fiit.stuba.sk/~pospichal/halanova/implementacia%20systemu/shell1.htm

[7] http://www.smnd.sk/lovasko/triediace_algoritmy.pdf

Prílohy

Príloha č.1

Zoznam obrázkov

Zoznam tabuliek

Zoznam grafov

Príloha č.1

Zoznam textových súborov

Textový súbor č. 1 - history.log

Príloha č. 3

Zdrojový kód programu

Zdrojový kód programu sa nachádza na priloženom CD v priečinku Zdrojový kód. Okrem zdrojového kódu programu sa na priloženom CD nachádza aj samotný program.