EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/B/2021/36124048426038532

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

Bakalárska práca

2021 **Michal Knor**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

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

Bakalárska práca

Študijný program: Hospodárska informatika
Študijný odbor: Hospodárska informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2021 **Michal Knor**

Čestné vyhlásenie

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

V Bratislave dňa 14.5.2021 .....

Podpis

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Knor
Študijný program: hospodárska informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: Bakalárska záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Porovnanie exekučnej efektívnosti triediacich algoritmov Heap 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 Heap Sort a Insertion sort. V rámci bakalárskej práce študent vytvorí program v jazyku C, v ktorom implementuje oba triediace algoritmy Heap Sort a Insertion Sort. Vytvorený program bude schopný zmerať exekučnú efektívnosť oboch triediacich algoritmov pri usporadúvaní veľkého súboru dát, napr. poľa s niekoľko stotisíc prvkami. Pomocou výstupov programu študent vykoná porovnanie exekučnej efektívnosti triediacich algoritmov Heap 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: 25.03.2020

Dátum schválenia: 26.03.2020

Ing. Mgr. Peter Schmidt, PhD.
vedúci katedry

ABSTRAKT

KNOR, Michal: *Porovnanie exekučnej efektívnosti triediacich algoritmov Heap 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, 2021, 59 strán.

Cieľom bakalárskej práce je vytvoriť program v programovacom jazyku C, ktorý je schopný usporiadať a zaznamenať exekučný čas triediacich algoritmov Heap Sort a Insertion Sort. Práca je rozdelená do 4 hlavných kapitol. Obsahuje 3 grafy, 2 tabuľky a 2 prílohy. Prvá kapitola je venovaná teoretickému popisu problematiky a vysvetleniu princípu vybraných triediacich algoritmov. V ďalšej časti sa charakterizuje cieľ práce a metodika. Záverečná kapitola sa zaoberá opisom programu a uskutočnenými meraniami. Výsledkom riešenia danej problematiky je vytvorenie programu v jazyku C, ktorý je schopný usporiadať a zaznamenať exekučný čas triediacich algoritmov Heap Sort a Insertion Sort.

Kľúčové slová: algoritmus, triedenie, Insertion Sort, Heap Sort, jazyk C

ABSTRAKT

KNOR, Michal: *A comparison of an execution efficiency of Heap Sort and Insertion Sort sorting algorithms that sort large data sets in a program created in C* – University of Economics Bratislava. Faculty of Business Informatics; Department of Applied Informatics. – Supervisor: Ing. Igor Košťál, PhD. – Bratislava: FHI, 2021, 59 pages.

The aim of the bachelor thesis is to create a program in the C programming language, which can sort and record the execution time of sorting algorithms Heap Sort and Insertion Sort. The work is divided into 4 main chapters. It contains 3 graphs, 2 tables and 2 appendices. The first chapter is devoted to a theoretical description of the problem and an explanation of the principle of selected sorting algorithms. The next part characterizes the aim of the work and methodology. The final chapter is about the description of the program and the measurements performed. The result of solving this problem is the creation of a program in C language, which is able to sort and record the execution time of sorting algorithms Heap Sort and Insertion Sort.

Keywords: algorithm, sorting, Insertion Sort, Heap Sort, C language

Obsah

Úvod 9

1 Súčasný stav riešenej problematiky doma a v zahraničí 10

1.1 Triedenie 10

1.2 Vnútorné a vonkajšie triedenie 10

1.3 Rekurzia a iterácia 10

1.4 Stabilita 11

1.5 Výpočtová zložitosť 12

1.6 Vybrané triediace algoritmy 13

1.6.1 Bubble Sort 13

1.6.2 Selection Sort 15

1.6.3 Quick Sort 18

1.6.4 Insertion Sort 20

1.6.5 Heap Sort 23

2. Cieľ práce 26

3. Metodika práce a metódy skúmania 27

4 Výsledky práce 28

4.1 Opis programu 28

4.1.1 Opis štruktúr 30

4.1.2 Opis funkcií 31

4.2 Funkčnosť programu 48

4.3 Výsledky meraní exekučných časov 50

4.3.1 Výsledky meraní exekučných časov na náhodných súboroch 50

4.3.2 Výsledky meraní exekučných časov na vybraných súboroch 52

4.3.3 Záver meraní 53

Záver 54

Zoznam použitej literatúry 55

Zoznam ilustrácií a zoznam tabuliek a grafov 56

Zoznam obrázkov 56

Zoznam tabuliek 57
Zoznam grafov 57
Prílohy 59

Úvod

Usporadúvanie je v bežnom živote veľmi dôležité. S usporiadanými prvkami vieme pracovať efektívnejšie a ďalšia práca s nimi je oveľa jednoduchšia. Najbežnejšie sa stretávame s abecedným usporiadaním, ktoré môžeme nájsť napríklad v slovníkoch, ale aj v menných zoznamoch.

Triedenie je v informatike takisto veľmi užitočné. Niekedy je vhodnejšie uchovávať ten istý súbor viackrát podľa iného usporiadania, ako je to napríklad v databázových systémoch.

Ak je počet prvkov, ktoré sa majú utriediť, malý, je častokrát výhodnejšie naprogramovať a použiť jednoduchý triediaci algoritmus. Avšak pri väčšom počte prvkov sa stáva tento triediaci algoritmus neefektívnym a musíme siahnuť na efektívnejší spôsob triedenia. [1]

Väčšina programovacích jazykov má zabudovanú funkciu, pomocou ktorej je možné utriediť pole. Programátor častokrát túto funkciu využije, čím ušetrí čas. Tému som si vybral preto, lebo aj ja sám radšej siahnem na vstavanú funkciu. Taktiež mi je ako programátorovi táto téma veľmi blízka.

1 Súčasný stav riešenej problematiky doma a v zahraničí

1.1 Triedenie

Pod triedením sa rozumie taký algoritmus, ktorý usporadúva vstupnú postupnosť prvkov (pole) podľa nejakého porovnania. Toto porovnanie môže usporiadať prvky vzostupne, zostupne, ale aj na základe nejakého algoritmu. Prvky sa môžu skladať z viacerých častí, pričom triedenie môžeme navrhnúť tak, že prvky utriedime iba podľa niektorých častí. [1, 2]

1.2 Vnútorné a vonkajšie triedenie

V prípade, že triedenie vykonávame iba na RAM pamäti, tak sa jedná o vnútorné triedenie. To je možné iba vtedy, ak dáta zo súboru, ktorý má algoritmus utriediť, sa zmestia do tejto pamäte. Algoritmus teda načíta dáta zo súboru do RAM pamäte, kde sa utriedia a následne sa takto utriedené dáta zapíšu do výsledného súboru. Pri vonkajšom triedení triediaci algoritmus pristupuje ku vstupnému súboru po častiach tak, aby sa zmestili do RAM pamäte. [1]

Existuje viacero spôsobov ako je možné naprogramovať vonkajšie triedenie. Jedným zo spôsobov, ako utriediť dáta vonkajším triedením, je dáta rozdeliť na viacero segmentov, ktoré sa zmestia do RAM pamäte a následne tieto segmenty triediaci algoritmus utriedi. Každý utriedený segment sa zapíše do nového dočasného súboru, ktorý sa vytvorí na pevnom disku zariadenia. Tieto vzniknuté súbory je potom možné utriediť do výsledného súboru tak, že sa vždy zoberie postupne prvý prvok z každého súboru a ten, ktorý je z týchto prvkov najmenší, sa zapíše do výsledného súboru, pričom tento prvok sa ďalej už neporovnáva. Po zapísaní posledného prvku z dočasného súboru, sa môže tento súbor vymazať. Takto sa postupuje, až kým nie sú všetky dáta vo výslednom súbore.

V tejto práci sme sa zaoberali iba vnútorným triedením.

1.3 Rekurzia a iterácia

Iterácia je časť algoritmu, ktorá sa vykonáva v slučke za nejakej podmienky. Rekurzia v programovaní je mechanizmus, keď funkcia volá samú seba. Najčastejšie sa používa na riešenie problémov, ktoré sa dajú rozdeliť na menšie časti. Medzi známe problémy, ktoré sú implementované rekurziou, patrí: vyhľadávanie do hĺbky, Hanojské veže, Fibonacciho čísla. Každú rekurziu je možné prepísať na iteráciu a naopak, avšak algoritmus sa môže stať horšie čitateľný. Nahradenie rekurzie iteráciou je výhodné, ak chceme dosiahnuť nižší exekučný čas, keďže v programovacom jazyku C (a aj v iných programovacích jazykoch) je rekurzia spravidla pomalšia ako iterácia. [3, 4]

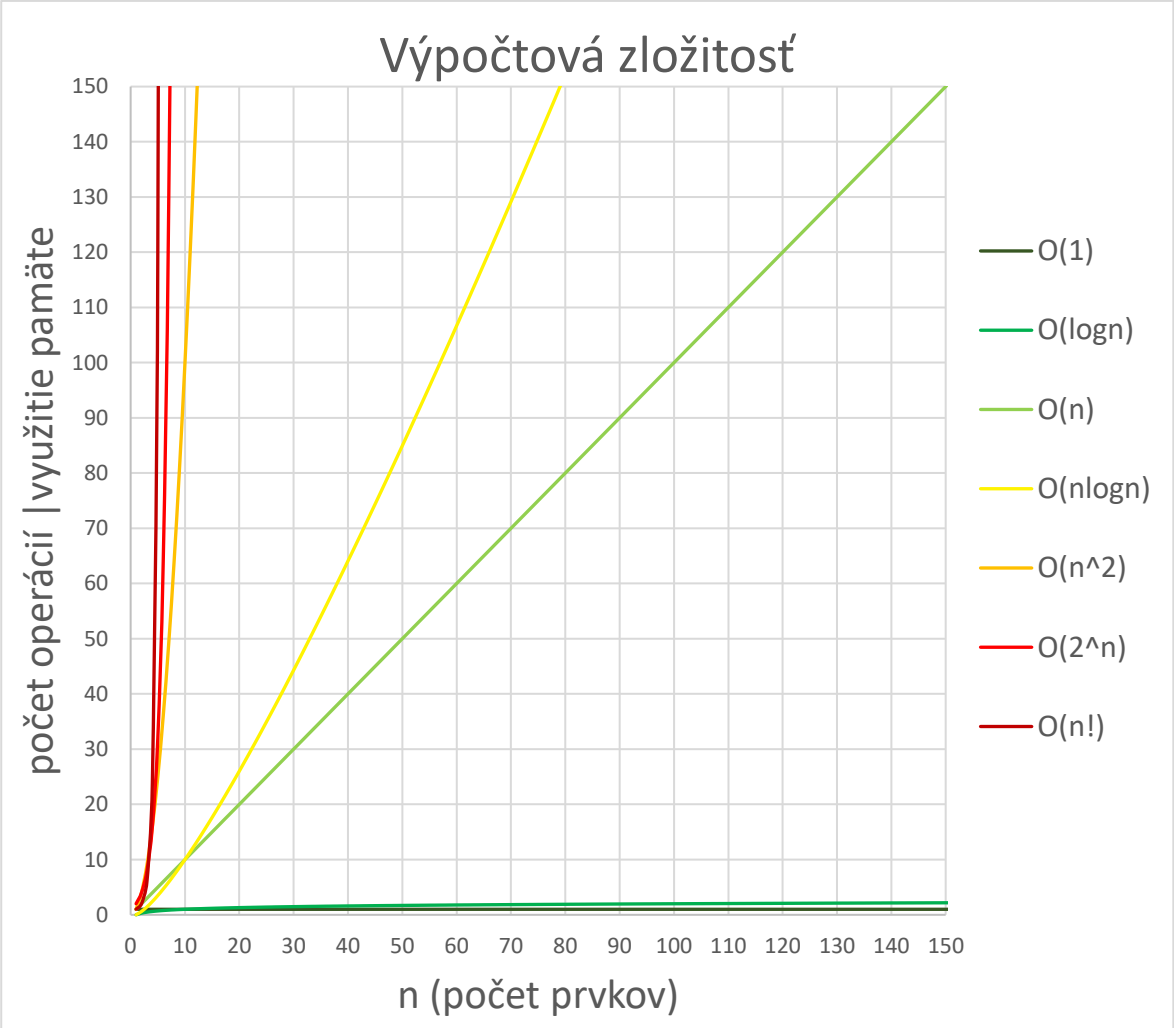
1.4 Stabilita

Triediace algoritmy môžu byť stabilné a nestabilné. Stabilita v triediacich algoritmoch nastáva vtedy, ak sa poradie rovnakých prvkov nezmení. Je dôležitá, ak vstupný súbor pozostáva z viacerých parametrov, pričom jeden z nich je už v správnom poradí. Napríklad ak by sme potrebovali utriediť pole, ktoré sa skladá z mena a z priezviska, pričom priezviská už máme zoradené a potrebujeme pole zoradiť podľa mena a následne podľa priezviska. Ak by sme na tomto poli spustili stabilný algoritmus, stačilo by nám porovnávať iba rozdiely mien a po utriedení by sme dostali želaný výsledok. Ak by sme ale spustili na tomto poli nestabilný algoritmus, ktorý by porovnával iba mená, tak by sme nemuseli dostať ako výsledok pole zoradené podľa mena a priezviska a teda pole by mohlo byť zoradené iba podľa mena a nie aj podľa priezviska. [5]

Nestabilné triediace algoritmy je možné prerobiť na stabilné. Algoritmus by sme mohli prerobiť tak, aby pri zhode primárnych prvkov poľa (mená), porovnával aj tie sekundárne (priezviská). V takomto prípade by ale algoritmus mohol byť veľmi pomalý, čo záleží od vstupného poľa. Vhodnejším spôsobom by bolo dočasne pridať na koniec každého mena aj jeho počiatočnú pozíciu alebo použiť asociatívne pole. Pri zhode mien, by sa porovnávala počiatočná pozícia. Tento spôsob je taktiež vhodný v prípade, že nevieme akým spôsobom sú zoradené sekundárne prvky poľa. Avšak v každom prípade by sa rýchlosť algoritmu zhoršila a zároveň by potreboval používať väčšiu časť pamäte.[5]

1.5 Výpočtová zložitosť

Výpočtová zložitosť vyjadruje ako náročný je výpočet daného algoritmu. Tá sa delí na časovú a pamäťovú zložitosť. Časovú zložitosť získame tak, že spočítame počet operácií vykonávaných algoritmom. Pri pamäťovej zložitosti záleží akú veľkosť pamäte algoritmus používa. Výpočtová zložitosť sa označuje symbolom O(f), pričom f je funkcia. Jej hlavným parametrom býva spravidla symbol n, ktorý určuje veľkosť vstupného súboru. [1, 6]



Graf č. 1: Grafické znázornenie výpočtovej zložitosti [Zdroj: vlastné spracovanie]

Všetky triediace algoritmy, ktoré spomenieme v nasledujúcej kapitole majú jednu z týchto zložitostí zobrazených na grafe č. 1, či už pamäťovú alebo časovú. Za rýchle triediace algoritmy sa považujú tie, ktoré majú priemernú časovú zložitosť menšiu alebo rovnú O(n log n). Časovú zložitosť O(1) pri triedení nikdy nedosiahneme, keďže samotná podmienka, či je pole utriedené má zložitosť O(n).

1.6 Vybrané triediace algoritmy

V tejto kapitole si vysvetlíme princíp vybraných triediacich algoritmov, pseudokód, výpočtovú zložitosť, použitie a ukážeme si aj vizualizáciu týchto algoritmov na rovnakom päťprvkovom poli. Farba podfarbenia v časti vizualizácie má takýto význam:

- Biela – s prvkom sa nič nedeje
- Modrá – táto farba sa používa vždy vo dvojici a zvýrazňuje, ktoré prvky sa porovnávajú
- Žltá – znázorňuje na ktorých miestach v poli nastala zmena oproti predchádzajúcemu stavu
- Zelená – prvok je utriedený

1.6.1 Bubble Sort

Bubble Sort patrí medzi stabilné a jednoducho implementovateľné triediace algoritmy. Opakovane vymieňa susedné prvky ak sú v zlom poradí. V prípade, že už nemá aké prvky vymeniť, algoritmus úspešne zotriedil vstupné pole. Menšie prvky takzvane „prebublinkujú“ vyššie, podľa čoho sa aj tak volá. [7]

1.6.1.1 Pseudokód

```
PROCEDURE bubbleSort(list_to_sort)
    swapped = true
    WHILE swapped == true
        swapped = false
        FOR i → 0 to length(list_to_sort) - 1
            IF list_to_sort[i] > list_to_sort[i + 1]
                swap(list_to_sort[i], list_to_sort[i + 1])
                swapped = true
            END IF
        END FOR
    END WHILE
END PROCEDURE
```

Obrázok č. 1: Pseudokód triediaceho algoritmu Bubble Sort [Zdroj: vlastné spracovanie]

1.6.1.2 Výpočtová zložitosť

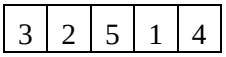
Časová zložitosť záleží od vstupného poľa. V najlepšom prípade, teda keď je pole už zotriedené, je časová zložitosť O(n). V tom najhoršom a zároveň aj priemernom je to $O(n^2)$, čím sa zaraďuje medzi pomalé algoritmy. Keďže si nepotrebuje uchovávať kópiu alebo časť triedeného poľa, má pamäťovú zložitosť O(1).

1.6.1.3 Použitie

Bubble Sort by sa nemal používať na utriedenie veľkých súborov, kvôli jeho časovej zložitosti. Avšak ak sú tieto súbory takmer utriedené a na ich úplné utriedenie stačí výmena pár susediacich prvkov, je Bubble Sort veľmi vhodný.

1.6.1.4 Vizualizácia

Vizualizácia triediaceho algoritmu Bubble Sort na nasledujúcom poli:

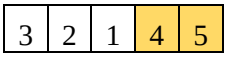


Algoritmus porovná postupne prvky poľa a vymení ich ak sú v nesprávnom poradí:

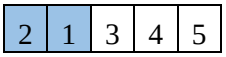
Prvky sú v zlom poradí, preto sa vymenia:

Pokračuje sa v porovnávaní zvyšných prvkov:

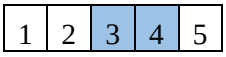
Prvky sú opäť v zlom poradí, takže sa vymenia:



Algoritmus sa dostal na koniec poľa a preto začne od začiatku a proces sa zopakuje:



Aj keď sú už prvky zoradené v správnom poradí, algoritmus sa ešte raz vykoná, keďže podmienkou pre jeho ukončenie je, že sa žiadne prvky nevymenili:



1.6.2 Selection Sort

Selection Sort je nestabilný a taktiež ľahko implementovateľný triediaci algoritmus. Opakovane hľadá najmenší prvok, ktorý ešte nebol zoradený a presunie ho na začiatok nezoradenej časti poľa (respektíve na koniec utriedenej časti poľa). Prvok, ktorý sa nachádzal na presunutom mieste sa vymení s najmenším nájdeným prvkom. Algoritmus si pamätá, ktorá časť poľa je už utriedená a ktorú časť ešte treba utriediť. [8]

1.6.2.1 Pseudokód

```
PROCEDURE selectionSort(list_to_sort)
        FOR j → 0 to length(list_to_sort) – 1
                min = j
                FOR i → j +1 to length(list_to_sort)
                        IF list_to_sort[i] > list_to_sort[min]
                                min = i
                        END IF
                END FOR
                swap(list_to_sort[min], list_to_sort[j])
        END FOR
END PROCEDURE
```

Obrázok č. 2: Pseudokód triediaceho algoritmu Selection Sort [Zdroj: vlastné spracovanie]

1.6.2.2 Výpočtová zložitosť

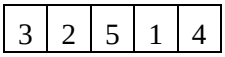
Čo sa týka časovej zložitosti, Selection Sort je veľmi pomalý. V každom prípade sa bude algoritmus vykonávať rovnaký počet krát, pričom časová zložitosť je vždy $O(n^2)$. Jediná výhoda tohto algoritmu (okrem jednoduchej implementácie) je, že má pamäťovú zložitosť O(1).

1.6.2.3 Použitie

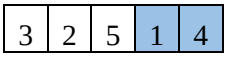
Selection Sort je jedným z najpomalších triediacich algoritmov. Nie je vhodné ho použiť na utriedenie väčších súborov. Na druhej strane je veľmi jednoduchý na pochopenie a teda ľahko naprogramovateľný. Taktiež je vhodné ho použiť ak sme limitovaný na veľkosť RAM pamäte, keďže hľadá postupne maximálne prvky a teda po ich nájdení ich môžeme rovno zapísať do výsledného súboru.

1.6.2.4 Vizualizácia

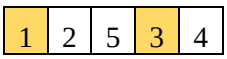
Vizualizácia triediaceho algoritmu Selection Sort na nasledujúcom poli:



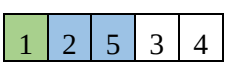
Algoritmus hľadá v celom poli najmenší prvok. Najprv si zapamätá hodnotu na prvom mieste a porovnáva ju so zvyšnými prvkami poľa. Ak nájde menší prvok, tak si zapamätá ten a porovnáva ho so zvyškom poľa, ktoré ešte neprešiel:

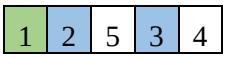


Našiel sa najmenší prvok poľa a ten sa vymení s prvým prvkom:



Pokračuje sa v hľadaní najmenšieho prvku v neutriedenej časti





Našiel sa najmenší prvok v neutriedenej časti poľa a ten sa vymení s prvým prvkom, respektíve druhým, teda v tomto prípade sa nič nestane:

Takýmto spôsobom sa pokračuje až kým algoritmus zoradí všetky prvky poľa:

1.6.3 Quick Sort

Quick Sort patrí medzi nestabilné triediace algoritmy. Algoritmus rozdelí prvky poľa do dvoch častí podľa zvoleného prvku. Tento prvok sa taktiež nazýva pivot. V jednej časti sú prvky menšie ako pivot a v druhej väčšie. Obe časti sa rekurzívne triedia rovnakým spôsobom až dokým nie je ich veľkosť rovná jednému prvku. Ak sa už ďalej tieto novovzniknuté časti nedajú rozdeliť, tak je pole utriedené. Jeho zaujímavosťou je to, že má rôzne typy implementácie výberu pivota, pričom jedna z nich je založená na náhode. Výber pivotu môže byť napríklad takýto: [9]

- prvý element
- posledný element
- medián
- náhodný prvok

1.6.3.1 Pseudokód

```
PROCEDURE partition(list_to_sort, lower_index, upper_index)
        pivot_index = (lower_index – 1)
        pivot = list_to_sort[upper_index]
        FOR j → lower_index to upper_index
                IF pivot > list_to_sort [j]
                        pivot_index = pivot_index + 1
                        swap(list_to_sort[pivot_index], list_to_sort[j])
                END IF
        END FOR
        swap(list_to_sort[pivot_index + 1], list_to_sort[upper_index])
        RETURN pivot_index + 1
END PROCEDURE

PROCEDURE quickSort(list_to_sort, lower_index, upper_index)
        IF lower_index < upper_index
pivot_index = partition(list_to_sort, from, to)
                quickSort(list_to_sort, lower_index, pivot _index – 1)
                quickSort(list_to_sort, pivot_index + 1, upper_index)
        END IF
END PROCEDURE
```

Obrázok č. 3: Pseudokód triediaceho algoritmu Quick Sort [Zdroj: vlastné spracovanie]

1.6.3.2 Výpočtová zložitosť

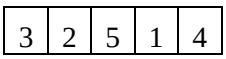
Časová zložitosť tohto algoritmu záleží od vstupného poľa. V prípade, že pivot rozdelí pole vždy na dve rovnaké časti, je časová zložitosť O(n log n). Naopak, ak je pole zoradené tak, že ho pivot rozdelí vždy na prázdne pole a zvyšnú časť poľa, tak je časová zložitosť $O(n^2)$. Priemerná časová zložitosť je O(n log n). Pamäťová zložitosť tohto Sortu je vždy O(log n). [9]

1.6.3.3 Použitie

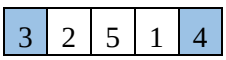
Quick Sort je jedným z najrýchlejších triediacich algoritmov a preto je vhodné ho použiť na utriedenie veľkých súborov. Jeho veľkou nevýhodou je, že vyžaduje viac miesta v RAM pamäti ako iné triediace algoritmy.

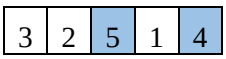
1.6.3.4 Vizualizácia

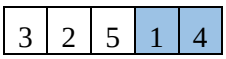
Vizualizácia triediaceho algoritmu Quick Sort, ktorý ako pivot vždy vyberá posledný prvok, na nasledujúcom poli:

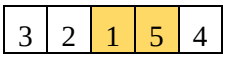


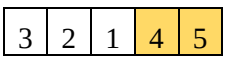
Ako pivot sa teda zvolí 4 a pole sa rozdelí na dve časti. Nasleduje rozdelenie poľa na dve časti podľa tohto pivotu:



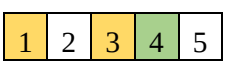




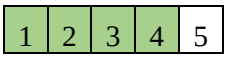




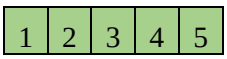
Pokračuje sa v triedení prvej časti poľa a ako pivot sa zvolí 1:



Vznikli ďalšie dve časti poľa, ale keďže majú iba jeden prvok (1 a 3), tak sú už utriedené:



Následne sa utriedi druhá časť poľa, ale keďže má tiež iba jeden prvok (5), tak už je utriedená. Algoritmus teda utriedil vstupné pole:



1.6.4 Insertion Sort

Insertion Sort patrí medzi stabilné a ľahko implementovateľné algoritmy. Postupne prechádza vstupné pole a prvky vloží na poprednejšie miesto, pokiaľ je prvok na tomto mieste väčší. Algoritmus prejde toto vstupné pole iba raz a následne je pole utriedené. Algoritmus sa volá Insertion, kvôli tomu, že prvky vkladá. [10]

1.6.4.1 Pseudokód

```
PROCEDURE insertionSort(list_to_sort)
        FOR i → 1 to length(list_to_sort)
                item_to_sort = list_to_sort[i]
                j = i
                WHILE j > 0 AND list_to_sort[j – 1] > item_to_sort
                        list_to_sort[j] = list_to_sort[j – 1]
                        j = j – 1
                END WHILE
        list_to_sort[j] = item_to_sort
        END FOR
END PROCEDURE
```

Obrázok č. 4: Pseudokód triediaceho algoritmu Insertion Sort [Zdroj: vlastné spracovanie]

1.6.4.2 Výpočtová zložitosť

Najlepšia časová zložitosť tohto algoritmu je O(n), čo nastáva v prípade, že je vstupné pole už utriedené. Najhoršia časová zložitosť ako aj priemerná sa uvádza $O(n^2)$. Pamäťová zložitosť tohto algoritmu je O(1), keďže algoritmus triedi prvky rovno na vstupnom poli a nepotrebuje si vytvárať kópiu alebo časť tohto poľa.

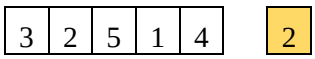
1.6.4.3 Použitie

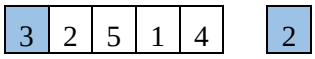
Tento triediaci algoritmus nie je vhodné použiť pre veľké súbory. Avšak, je vhodné ho použiť aj pri veľkých súboroch, ak je iba pár prvkov na zlých pozíciách.

1.6.4.4 Vizualizácia

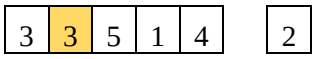
Vizualizácia triediaceho algoritmu Insertion Sort, na nasledujúcom poli:

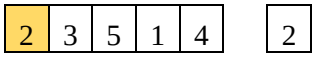
Algoritmus postupne prechádza prvky poľa a porovnáva ich, pričom začína na druhom prvku:



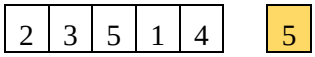


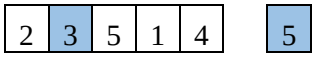
Prvok 2 je na zlom mieste, keďže 3 je väčšia. 2 sa vloží do pomocnej pamäte. Následne sa zisťuje, či poprednejšie prvky sú väčšie od 2, ale keďže sa pred 3 nenachádza žiaden prvok, tak sa najprv 3 zapíše na miesto 2 a následne sa 2 zapíše na prvú pozíciu:



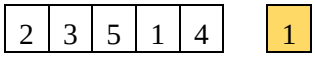


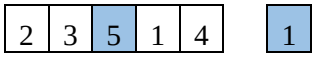
Pokračuje sa v ďalšom prvku:



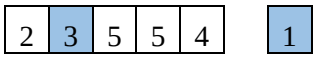


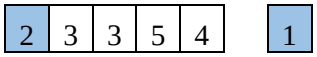
5 nie je menšia ako 3 a preto sa pokračuje ďalej:

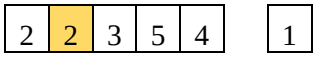


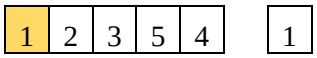


1 je menšia ako 5 a preto sa 1 vloží do pomocnej pamäte a postupne sa porovnáva s predchádzajúcimi prvkami a vloží ju na správne miesto:

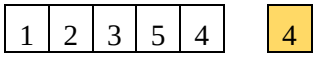








Nakoniec sa utriedi ešte posledný prvok:



1.6.5 Heap Sort

Heap Sort je nestabilný a ťažšie implementovateľný triediaci algoritmus. Prvky uchováva v halde. Halda je typ binárneho stromu, pre ktorú platí, že má minimálnu hĺbku. Rodičovský prvok má nanajvýš dvoch potomkov a potomok je menší alebo rovný ako jeho rodič. Prvky sa pritom vkladajú zľava. Haldu je možné realizovať aj na vstupnom poli, bez nutnosti vytvorenia pomocného poľa (alebo inej dátovej štruktúry). V takomto prípade platí, že ak sa indexuje od 0, tak: [11, 12]

- ľavý potomok = 2 * *aktuálny_index* + 1
- pravý potomok = 2 * *aktuálny_index* + 2
- rodič = (*aktuálny_index* – 1) / 2

Heap Sort vykonáva triedenie na poli tak, že najprv vytvorí z poľa haldu. Následne vymení posledný prvok s prvým a prvky preusporiada, aby pre pole platili podmienky haldy. Tento posledný prvok sa stáva utriedeným, keďže je najväčším prvom z celého poľa. Proces sa opakuje až kým sa neusporiadali všetky prvky poľa. [7, 8]

1.6.5.1 Pseudokód

```
PROCEDURE heapify(list_to_sort, parentIndex, heapLength)
        largest = parentIndex
        left = 2 * parentIndex + 1
        right = 2 * parentIndex + 2
        IF left < heapLength AND list_to_sort[left] > list_to_sort[largest]
                largest = left
        END IF
        IF right < heapLength AND list_to_sort[right] > list_to_sort[largest]
                largest = right
        END IF
        IF largest != parentIndex
                swap(list_to_sort[parentIndex], list_to_sort[largest])
                heapify(list_to_sort, largestIndex, heapLength)
        END IF
END PROCEDURE

PROCEDURE heapSort(list_to_sort)
        FOR i → length(list_to_sort) / 2 downto 0
                heapify(list_to_sort, i – 1, length(list_to_sort))
        END FOR
FOR i → length(list_to_sort) downto 0
        swap(list_to_sort[0], list_to_sort[i – 1])
                heapify(list_to_sort, 0, i – 1)
        END FOR
END PROCEDURE
```

Obrázok č. 5: Pseudokód triediaceho algoritmu Heap Sort [Zdroj: vlastné spracovanie]

1.6.5.2 Výpočtová zložitosť

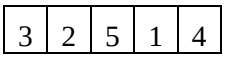
Časová zložitosť tohto algoritmu je vždy O(n log n), čím sa zaraďuje medzi najrýchlejšie triediace algoritmy. Pamäťová zložitosť tohto algoritmu je O(1), v prípade, že haldu vytvorí na vstupnom poli.

1.6.5.3 Použitie

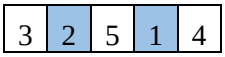
Heap Sort je jedným z najefektívnejších triediacich algoritmov a preto sa používa na utriedenie veľkých súborov. Jeho výhodou je ako aj pamäťová zložitosť, tak aj časová. Keďže časová zložitosť je vždy rovnaká, bez ohľadu na vstupné pole, tak je jedinou jeho nevýhodou, že môže byť pri vybraných súboroch pomalší ako iné triediace algoritmy.

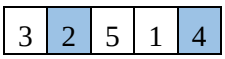
1.6.5.4 Vizualizácia

Vizualizácia triediaceho algoritmu Heap Sort, na nasledujúcom poli:

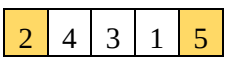


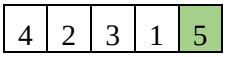
Najprv sa prvky v poli preusporiadajú, aby bolo pole haldou. V prípade, že obaja potomkovia sú väčší ako jeho rodič, vymení sa s rodičom ten potomok, ktorý je väčší:



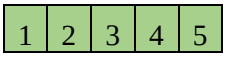


Prvok na prvom mieste (5) je najväčší z celej neutriedenej časti a vymení sa s posledným neutriedeným prvkom (2). Prvok 5 je teda utriedený. Pole sa opäť prerobí na haldu, avšak iba z neutriedenej časti:





Prvky 4 a 1 sa vymenia a celý proces sa opakuje až kým sa neutriedi celé pole:



2. Cieľ práce

Cieľom bakalárskej práce bolo vytvoriť program pomocou ktorého by sme vedeli porovnať exekučný čas triediacich algoritmov Insertion Sort a Heap Sort.. Triedenie by sa dalo vykonať buď na vybranom súbore, ktorý používateľ zadá alebo je možné tiež súbor vygenerovať na základe určitých parametrov a to počet riadkov súboru a maximálny počet znakov riadku. Takto vygenerovaný súbor by mal obsahovať náhodné znaky malých písmen latinskej abecedy. Triedenie, ktoré sa na súboroch vykoná by malo byť zapísané do textového súboru, pomocou ktorého by sme mali vedieť porovnať exekučný čas triediacich algoritmov. Výsledkom by mala konzolová aplikácia ovládateľná používateľom, pričom by boli splnené všetky spomenuté požiadavky.

3. Metodika práce a metódy skúmania

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

Vývoj programu a následné porovnanie triediacich algoritmov bolo vykonané na notebooku Acer Aspire 5 A515-51G s parametrami:

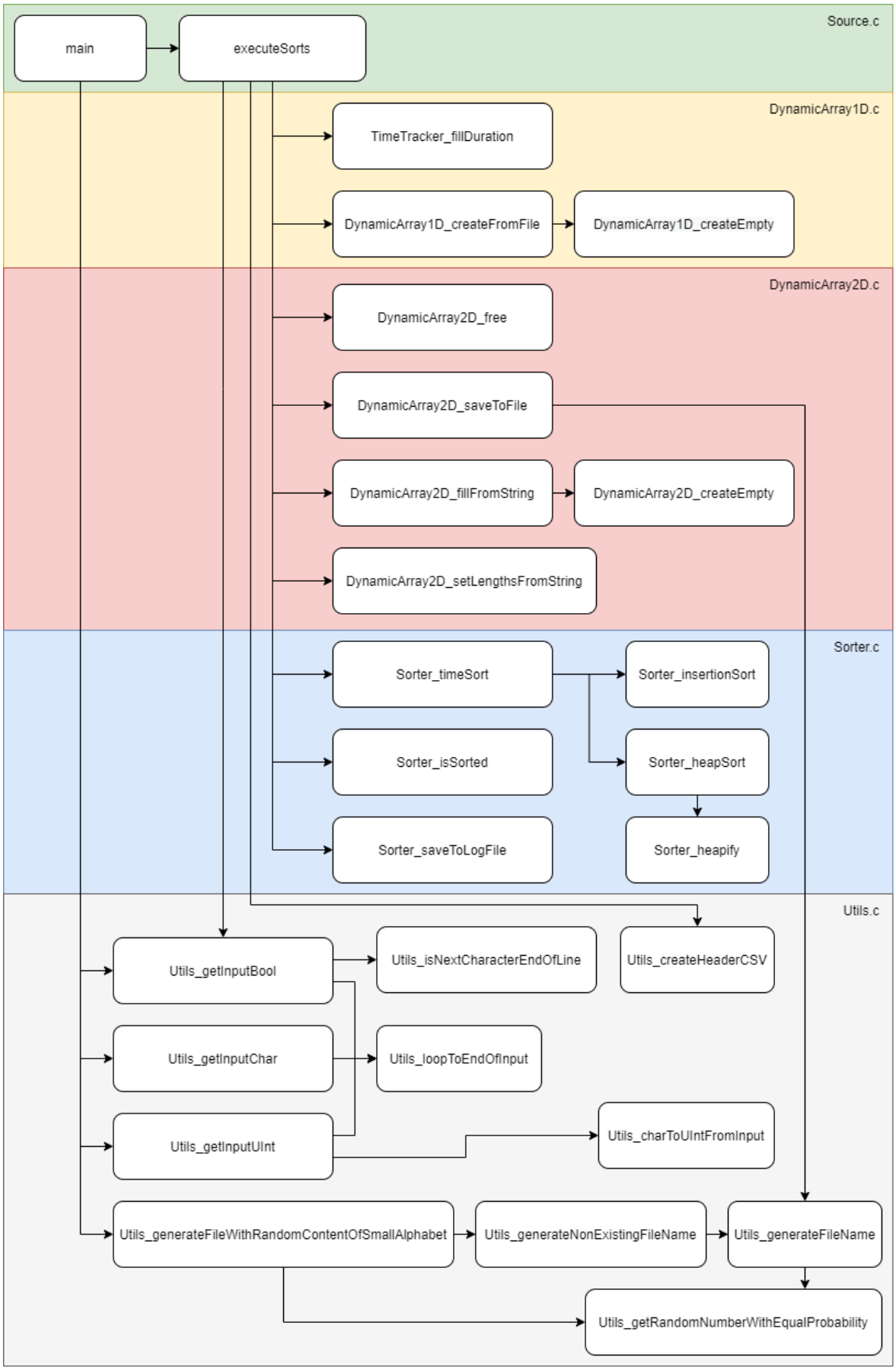
- Procesor: Intel® Core™ i5-8250U 1.60GHz (Turbo Boost 3.40GHz)
- RAM: 8GB
- Typ systému: 64-bitový operačný systém, Windows 10

Program bol vytvorený v programovacom prostredí Visual Studio.

4 Výsledky práce

4.1 Opis programu

V tejto kapitole si ukážeme ako program funguje a vysvetlíme jednotlivé funkcie programu. Program pozostáva zo štyroch vytvorených štruktúr a dvadsiatich siedmych funkcií, pričom všetky sú potrebné na správnu funkčnosť programu.



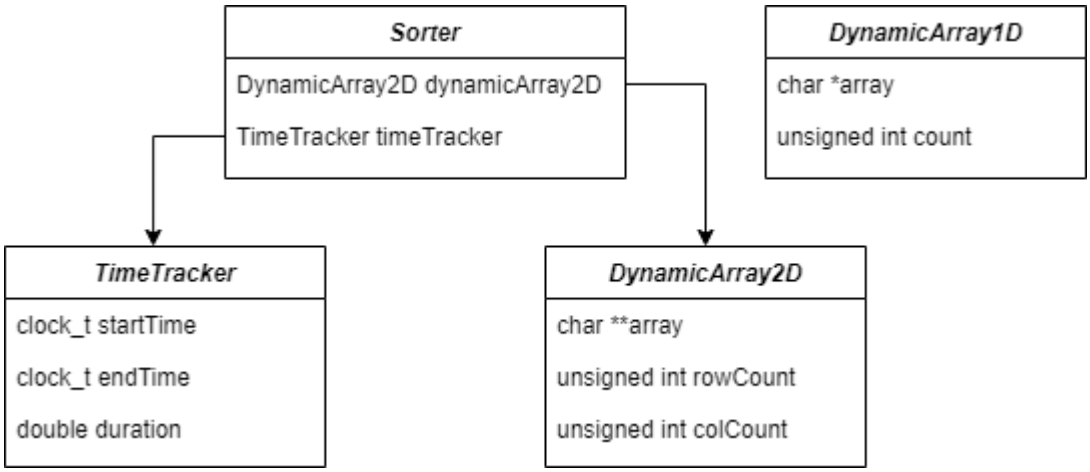
Obrázok č. 6: Mapa volaní funkcií [Zdroj: vlastné spracovanie]

Na obrázku č. 6 môžeme vidieť ako sa funkcie medzi sebou volajú a zároveň v akých súboroch sú umiestnené. Najprv si vysvetlíme prečo sú potrebné dané vytvorené štruktúry

a následne si opíšeme funkcie programu. Pri vysvetľovaní jednotlivých funkcií budeme postupovať po porade podľa súborov v ktorých sa funkcie nachádzajú, pričom niektoré funkcie spomenieme detailnejšie.

4.1.1 Opis štruktúr

Ako sme už spomenuli, program pozostáva zo štyroch doprogramovaných štruktúr, ktorých prepojenie môžeme vidieť na nasledujúcom obrázku:



Obrázok č. 7: Diagram štruktúr [Zdroj: vlastné spracovanie]

Štruktúry sme vytvorili na to, aby sa program stal čitateľnejším, ale najmä na enkapsuláciu, vďaka čomu teda nebudeme musieť uvádzať ako parametre funkcií všetky údaje, ktoré potrebujeme použiť. Tie sa nachádzajú v danej štruktúre a stačí nám poslať iba smerník na štruktúru. V nasledujúcich kapitolách si bližšie vysvetlíme jednotlivé premenné týchto štruktúr.

4.1.1.1 TimeTracker

Táto štruktúra nám slúžila na meranie času a je potrebná na to, aby sme vedeli odmerať čas triediacich algoritmov Insertion Sort a Heap Sort. Pozostáva z týchto premenných:

- *clock_t startTime* – slúži na uloženie začiatočného času
- *clock_t endTime* – slúži na uloženie konečného času
- *double duration* – do tejto premennej sa uloží trvanie času v sekundách, ktoré sa vypočíta z premenných *startTime* a *endTime*

4.1.1.2 DynamicArray1D

Ako už vyplýva z názvu, štruktúra slúži na uloženie dynamického jednorozmerného poľa. To znamená, že veľkosť poľa, ktoré má program alokovať, je pred spustením programu

neznáma. Na uloženie informácií o dynamickom poli si štruktúra musí pamätať tieto dve premenné:

- *char *array* – ukazovateľ na prvok poľa typu charakter
- *unsigned int count* – premenná, ktorá slúži na zaznamenanie veľkosti poľa

4.1.1.3 DynamicArray2D

Štruktúra uchováva dynamické dvojrozmerné pole. To znamená, že na rozdiel od jednorozmerného poľa, do dvojrozmerného pristupuje cez dva indexy. Prvý index určuje k akému riadku sa má pristupovať a druhý index ku stĺpcu. Štruktúra preto pozostáva z nasledovných premenných:

- *char **array* – ukazovateľ na ukazovateľ na prvok typu charakter
- *unsigned int rowCount* – počet riadkov poľa (prvý rozmer)
- *unsigned int colCount* – počet stĺpcov poľa (druhý rozmer)

4.1.1.4 Sorter

V tejto štruktúre si uchovávame informácie o štruktúrach DynamicArray2D a *TimeTracker*. Slúži nám teda na prepojenie týchto dvoch štruktúr, kde si ukladáme dĺžku trvania jednotlivých triediacich algoritmov na dynamickom dvojrozmernom poli. Ako sme už spomenuli, skladá sa z týchto dvoch premenných:

- *DynamicArray2D dynamicArray2D*
- *TimeTracker timeTracker*

4.1.2 Opis funkcií

V nasledujúcich kapitolách si vysvetlíme princíp väčšiny funkcií podľa súborov, v ktorých sa nachádzajú. To ako sa medzi sebou volajú sme si už zobrazili na mape volaní funkcií na obrázku č. 6.

4.1.2.1 Utils.c

V tomto súbore sme si vytvorili funkcie, ktoré nepatria k žiadnym štruktúram, teda nemajú ako parameter ukazovateľ na štruktúru. Niektoré z nich si bližšie vysvetlíme.

```
unsigned int Utils_getRandomNumberWithEqualProbability(unsigned int from, unsigned
int to) {
	unsigned int endingNumberInterval = RAND_MAX - RAND_MAX % (to - from + 1);
	unsigned int generatedNumber = rand();
	while (endingNumberInterval <= generatedNumber) {
		generatedNumber = rand();
	}
	generatedNumber = generatedNumber % (to - from + 1) + from;

	return generatedNumber;
}
```

Obrázok č. 8: Zdrojový kód funkcie *Utils_getRandomNumberWithEqualProbability*

[Zdroj: vlastné spracovanie]

Funkcia slúži na vygenerovanie náhodného čísla z intervalu <*from, to*>. Návratovou hodnotou tejto funkcie je toto vygenerované číslo. Na vygenerovanie náhodného čísla sme použili funkciu *rand()*, ktorá je súčasťou knižnice *stdlib.h*. Táto funkcia vygeneruje pseudo-náhodné číslo od 0 do *RAND_MAX*, pričom na našom počítači to je 32767. Následne toto vygenerované číslo vydelí rozdielom horného a dolného intervalu, vďaka čomu dostaneme náhodné číslo z tohto intervalu.

Pri väčšom intervale by číslo až tak náhodné nebolo. Ak by sme napríklad zvolili interval <0, 9999>, tak by bola väčšia šanca, že dostaneme čísla od 0 do 2767 ako čísla od 2768 do 9999, pretože hodnota 9999 nie je deliteľom hodnoty premennej RAND_MAX s hodnotou 32767. Na to, aby každé číslo z intervalu malo rovnakú šancu byť náhodne vygenerované, sme sa rozhodli pri takomto probléme, vygenerovať číslo až pokiaľ nie je číslo mimo náhodnej distribúcie. To znamená, že pri takto zvolenom intervale, ak by sa vygenerovalo číslo väčšie alebo rovné 30000 (najväčšia možná hodnota dosiahnutá celočíselným násobkom z rozdielu intervalov + 1), tak by sa vygenerovalo ešte raz až pokiaľ by sme nedostali číslo menšie ako 30000. Síce sme tým zvýšili priemerný čas potrebný na vygenerovanie jedného prvku, avšak prvky z daného intervalu majú približne rovnakú pravdepodobnosť byť vygenerované.

```
void Utils_generateFileName(char fileName[], char fileFolder[]) {
	struct tm tm;
	time_t now = time(0);
	localtime_s(&tm, &now);
	sprintf(fileName, "%s%04d-%02d-%02d %02d-%02d-%02d %04d.txt", fileFolder,
1900 + tm.tm_year, tm.tm_mon, tm.tm_mday, tm.tm_hour, tm.tm_min, tm.tm_sec,
Utils_getRandomNumberWithEqualProbability(0, 9999));
}
```

Obrázok č. 9: Zdrojový kód funkcie *Utils_generateFileName*

[Zdroj: vlastné spracovanie]

Táto funkcia nám slúži na vytvorenie názvu súboru, ktorý sa uloží do vstupnej premennej *fileName*. Na začiatok tejto premennej sa zároveň priradí aj priečinok, v ktorom sa má súbor nachádzať (parameter *fileFolder*). Obe tieto premenné sú statické polia znakov. Názov tohto súboru má formát: YYYY-MM-DD HH-NN-SS RRRR, pričom význam týchto písmen je nasledovný:

- YYYY – aktuálny rok
- MM – aktuálny mesiac
- DD – aktuálny deň
- HH – aktuálna hodina
- NN – aktuálna minúta
- SS – aktuálna sekunda
- RRRR – náhodné číslo z intervalu <0, 9999>

```
void Utils_generateNonExistingFileName(char fileName[], char fileFolder[]) {
	mkdir(fileFolder, 0777);
	Utils_generateFileName(fileName, fileFolder);
	FILE *file = fopen(fileName, "r");
	while (file) {
		fclose(file);
		Utils_generateFileName(fileName, fileFolder);
		file = fopen(fileName, "r");
	}
}
```

Obrázok č. 10: Zdrojový kód funkcie *Utils_generateNonExistingFileName*

[Zdroj: vlastné spracovanie]

Funkcia slúži na vygenerovanie názvu súboru, ktorý ešte neexistuje. Funkcia má dva parametre pričom obidva sú polia znakov a to *fileName* (názov súboru) a *fileFolder* (názov priečinka). Najprv funkcia vytvorí priečinok *fileFolder* do ktorého budeme ukladať súbor. Následne generuje pomocou funkcie *Utils_generateFileName* názvy súboru. To či súbor existuje zistíme pomocou funkcie *fopen*. Ak súbor existuje, tak sa do premennej *file* uloží smerník na štruktúru FILE. Avšak ak súbor neexistuje, tak sa do premennej uloží hodnota NULL. Generáciu mena súboru a následné otváranie sme preto obalili do podmieneného cyklu, pričom podmienka je iba samotný súbor. Tým pádom sme zaručili, že sa názov súboru bude generovať až dovtedy, pokiaľ súbor s týmto názvom ešte neexistuje v danom priečinku. Tento názov súboru zároveň funkcia ukladá do vstupnej premennej *fileName*, teda funkcia nepotrebuje mať žiadnu návratovú hodnotu a preto je typu *void*.

```
void Utils_generateFileWithRandomContentOfSmallAlphabet(unsigned int rowCount,
unsigned int colCount, char fileName[]) {
	Utils_generateNonExistingFileName(fileName, INPUT_FILE_PATH);
	FILE *file = fopen(fileName, "w");
	char character;
	for (unsigned int i = 0; i < rowCount; i++) {
		for (unsigned int j = 0; j < colCount - 1; j++) {
			character = Utils_getRandomNumberWithEqualProbability(96, 122);
			if (character == 96) {
				break;
			}
			fputc(character, file);
		}
		if (i + 1 < rowCount) {
			fputc('\n', file);
		}
	}
	fclose(file);
}
```

Obrázok č. 11: Zdrojový kód funkcie

U*tils_generateFileWithRandomContentOfSmallAlphabet* [Zdroj: vlastné spracovanie]

Táto funkcia slúži na naplnenie súboru pričom jeho názov je vygenerovaný pomocou funkcie *Utils_generateNonExistingFileName*. Taktiež funkcia tento súbor naplní náhodnými písmenami malej latinskej abecedy. Počet riadkov záleží od parametra *rowCount* a počet znakov týchto riadkov je rovný alebo menší parametru *colCount*. Posledným znakom každého riadku okrem posledného je ukončovací znak riadku '\n'. Funkcia nám slúžila na vygenerovanie súboru s náhodne vygenerovaným obsahom, ktorý sme mohli následne utriediť.

Ostatné funkcie nachádzajúce sa v tomto súbore nie sú až tak zaujímavé a netýkajú sa témy tejto bakalárskej práce, preto si ich iba jednoducho opíšeme:

```
void Utils_createHeaderCSV();

unsigned int Utils_isNextCharacterEndOfLine(char symbol[]);

void Utils_loopToEndOfInput(char symbol[]);

unsigned int Utils_charToUIntFromInput(char charNumber[], unsigned int charLength,
unsigned int *number);

unsigned int Utils_getInputBool(char text[]);

void Utils_getInputChar(char charArray[], unsigned int charLength, char text[]);

unsigned int Utils_getInputUInt(char text[], unsigned int minNumber, unsigned int
maxNumber);
```

Obrázok č. 12: Zdrojový kód hlavičiek ostatných funkcií v súbore *Utils.h*

[Zdroj: vlastné spracovanie]

Prvá funkcia slúži na vytvorenie hlavičky CSV súboru, v prípade že súbor neexistuje. Tento súbor nám slúžil na uloženie dĺžky trvania jednotlivých triediacich algoritmov, čo nám uľahčilo vytváranie grafov, ktoré sú zobrazené v neskorších častiach tejto práce. Ostatné funkcie slúžili na správne fungovanie konzolovej aplikácie, pomocou ktorých sa spracúvajú vstupy zadané používateľom.

4.1.2.2 TimeTracker.c

V tomto súbore sa nachádzajú všetky funkcie, ktoré majú prvý parameter typu smerníku na *TimeTracker*. Nachádza sa tu iba jedna funkcia

```
void TimeTracker_fillDuration(TimeTracker *this) {
	double duration = this->endTime - this->startTime;
	duration /= CLOCKS_PER_SEC;
	this->duration = duration;
}
```

Obrázok č. 13: Zdrojový kód funkcie *TimeTracker_fillDuration*

[Zdroj: vlastné spracovanie]

Táto funkcia slúži na vyplnenie času trvania, ktorý ubehol medzi začiatkom (*startTime*) a koncom (*endTime*) merania. Keďže tieto časy sú uvedené v počte tikov, tak sme potrebovali tento počet ešte vydeliť počtom tikov za sekundu, aby sme dostali počet sekúnd. Táto funkcia je veľmi dôležitá, keďže vďaka nej sme zistili trvanie jednotlivých triediacich algoritmov.

4.1.2.3 DynamicArray1D.c

V tomto súbore sa nachádzajú funkcie, ktoré pracujú so štruktúrou DynamicArray1D. Nachádzajú sa tu dve funkcie, ktoré sú potrebné na vytvorenie a naplnenie tejto štruktúry.

```
void DynamicArray1D_createEmpty(DynamicArray1D *this, unsigned int count) {
	this->count = count;
	this->array = (char *)malloc(count * sizeof(char));
}
```

Obrázok č. 14: Zdrojový kód funkcie *DynamicArray1D_createEmpty*

[Zdroj: vlastné spracovanie]

V tejto funkcií sa alokuje pamäť jednorozmerného poľa charakterov, ktorého veľkosť sa rovná vstupnému parametru *count*. K tomuto poľu vieme pristupovať cez ukazovateľ na štruktúru DynamicArray1D, pričom sa uchová aj premenná vyjadrujúca počet prvkov v poli.

```
void DynamicArray1D_createFromFile(DynamicArray1D *this, char filePath[]) {
	FILE *file = fopen(filePath, "r");
	int count = 1;
	char character;
	for (character = fgetc(file); character != EOF; character = fgetc(file)) {
		count = count + 1;
	}
	DynamicArray1D_createEmpty(this, count);
	fseek(file, 0, SEEK_SET);
	for (unsigned int i = 0; i < this->count-1; i++) {
		this->array[i] = fgetc(file);
	}
	this->array[count-1] = '\0';
	fclose(file);
}
```

Obrázok č. 15: Zdrojový kód funkcie *DynamicArray1D_createFromFile*

[Zdroj: vlastné spracovanie]

Uvedená funkcia slúži na prečítanie obsahu súboru a následné uloženie do dynamického jednorozmerného poľa. Cesta k súboru je uložená vo vstupnom parametri *filePath*. Táto cesta pritom môže byť aj absolútna a aj relatívna, keďže funkcia *fopen* vie pracovať s oboma typmi ciest. Tento súbor sa prečíta dvakrát. Prvým prečítaním sa zistí počet znakov súboru, podľa čoho vieme alokovať potrebne veľké pole. Následne sa zavolá funkcia na vytvorenie prázdneho poľa. Pri druhom čítaní sa obsah súboru už zapisuje do tohto poľa. Ako posledný prvok poľa sa zapíše znak '\0' označujúci ukončovací znak reťazca.

4.1.2.4 DynamicArray2D.c

Takisto ako aj v predchádzajúcej časti, v tomto súbore sa nachádzajú funkcie, ktoré súvisia so štruktúrou DynamicArray2D.

```
void DynamicArray2D_createEmpty(DynamicArray2D *this, unsigned int rowCount,
unsigned int colCount) {
	this->array = (char **)malloc(rowCount * sizeof(char *));
	for (unsigned int i = 0; i < rowCount; i++) {
		this->array[i] = (char *)malloc(colCount * sizeof(char));
	}
	this->rowCount = rowCount;
	this->colCount = colCount;
}
```

Obrázok č. 16: Zdrojový kód funkcie *DynamicArray2D_createEmpty*

[Zdroj: vlastné spracovanie]

Pomocou tejto funkcie sme vytvárali dvojrozmerné pole veľkosti *rowCount* * *colCount*. Funkcia vytvára v cykle jednorozmerné polia o veľkosti *colCount* pričom ich počet je rovný parametru *rowCount*. Následne sa adresy týchto polí uložia do prvkov poľa *array*, čím sme dostali dvojrozmerné dynamické pole.

```
void DynamicArray2D_free(DynamicArray2D *this) {
	for (unsigned int i = 0; i < this->rowCount; i++) {
		free(this->array[i]);
	}
}
```

Obrázok č. 17: Zdrojový kód funkcie *DynamicArray2D_free* [Zdroj: vlastné spracovanie]

Vo funkcií sa uvoľní z pamäte pole, ktoré sa alokovalo. Keďže v programovacom jazyku C nemáme garbage collector, tak sme museli pole uvoľňovať manuálne. Ak by sme tak nespravili, časom by sme si mohli zaplniť celú RAM pamäť a aplikácia by nám „spadla“.

```
void DynamicArray2D_setLengthsFromString(DynamicArray2D *this, DynamicArray1D
*string, char separator) {
	unsigned int counter = 1, colCount = 0, rowCount = 0;
	for (unsigned int i = 0; i < string->count; i++) {
		if (string->array[i] == separator || string->array[i] == '\0') {
			if (counter > colCount) {
				colCount = counter;
			}
			rowCount++;
			counter = 1;
		}
		else {
			counter++;
		}
	}
	this->colCount = colCount;
	this->rowCount = rowCount;
}
```

Obrázok č. 18: Zdrojový kód funkcie *DynamicArray2D_setLengthsFromString* [Zdroj: vlastné spracovanie]

V tejto funkcií sa nastavujú parametre štruktúry DynamicArray2D, ktoré vyjadrujú rozmery dynamického dvojrozmerného poľa (*colCount* a *rowCount*) podľa vstupného parametru *string*, ktorý je ukazovateľom na štruktúru DynamicArray1D. Na to aby sme z jednorozmerného poľa dostali dvojrozmerné, sme museli zároveň použiť parameter *separator*, ktorý vyjadruje podľa akého znaku sa má rozdeliť jednorozmerné pole. Keďže v dynamickom jednorozmernom poli sa nachádza vždy obsah načítaného súboru, tak sme ako oddeľovač používali vždy znak vyjadrujúci koniec riadku ‘\n’. Funkcia teda zistí zo vstupného poľa *string* najdlhšiu sekvenciu znakov medzi oddeľovačmi, ktorá sa zapíše do parametru *colCount* štruktúry DynamicArray2D.

```
void DynamicArray2D_fillFromString(DynamicArray2D *this, DynamicArray1D *string,
char separator) {
	DynamicArray2D_createEmpty(this, this->rowCount, this->colCount);
	unsigned int r = 0, c = 0, i;
	for (i = 0; i < string->count; i++) {
		if (string->array[i] == separator || string->array[i] == '\0') {
			this->array[r][c] = '\0';
			c = 0;
			r++;
			continue;
		}
		this->array[r][c] = string->array[i];
		c++;
	}
	if (c > 0) {
		this->array[r][c] = '\0';
	}
}
```

Obrázok č. 19: Zdrojový kód funkcie *DynamicArray2D_fillFromString*

[Zdroj: vlastné spracovanie]

Funkcia napĺňa dvojrozmerné pole, ktoré sa vytvorí hneď v prvom riadku kódu, pri volaní funkcie *DynamicArray2D_createEmpty*. Keďže po tomto volaní má už funkcia alokované dvojrozmerné pole, môže ho naplniť. Tak ako aj v predchádzajúcej funkcií to robí pomocou parametra *separator*, avšak v tejto funkcií sa pole napĺňa. Pole sa postupne napĺňa zo vstupného reťazca, pričom sa zvyšuje druhý index dvojrozmerného poľa o jeden a zároveň index reťazca. Ak sa prvok reťazca rovná znaku *separator*, tak sa zvýši prvý index dvojrozmerného poľa o jeden a druhý index sa nastaví na hodnotu 0. Takto sa postupne napĺňa dvojrozmerné pole, pričom ako posledný prvok jednotlivých polí sa zapíše ukončovací znak reťazcu ‚\n'.

```
void DynamicArray2D_saveToFile(DynamicArray2D *this, char fileName[]) {
	Utils_generateNonExistingFileName(fileName, SORTED_FILE_PATH);
	FILE *file = fopen(fileName, "w");
	for (unsigned int i = 0; i < this->rowCount - 1; i++) {
		char *test = this->array[i];
		fprintf(file, "%s\n", this->array[i]);
	}
	if (this->rowCount > 0) {
		fprintf(file, "%s", this->array[this->rowCount - 1]);
	}
	fclose(file);
}
```

Obrázok č. 20: Zdrojový kód funkcie *DynamicArray2D_saveToFile*

[Zdroj: vlastné spracovanie]

Táto funkcia uloží obsah poľa štruktúry DynamicArray2D do súboru pričom názov tohto súboru sa uloží do vstupnej premennej *fileName*. Táto funkcia používa funkciu *Utils_generateNonExistingFileName* pomocou ktorej sa vygeneruje názov súboru, ktorý

ešte neexistuje v danom priečinku, ktorý sa dá nastaviť v literály *SORTED_FILE_PATH*. Následne sa prechádza prvá dimenzia dvojrozmerného poľa a uloží sa jej obsah do výsledného súboru, pričom na jeho koniec sa vždy zapíše ukončovací znak riadku '\n'. Táto funkcia sa používa pri zapisovaní do súboru po vykonaní triediacich algoritmov na dvojrozmernom poli.

4.1.2.5 Sorter.c

V súbore *Sorter*.c sa nachádzajú funkcie, ktoré pracujú so vstupným parametrom typu smerník na štruktúru *Sorter*. Nachádzajú sa tu jednotlivé triediace algoritmy ako aj validácia či je pole utriedené a zapisovanie logovacieho súboru.

```
unsigned short int Sorter_isSorted(Sorter *this) {
	for (unsigned int i = 1; i < this->dynamicArray2D.rowCount; i++) {
		if (strcmp(this->dynamicArray2D.array[i-1], this-
>dynamicArray2D.array[i]) > 0) {
			return 0;
		}
	}
	return 1;
}
```

Obrázok č. 21: Zdrojový kód funkcie *Sorter_isSorted* [Zdroj: vlastné spracovanie]

Táto funkcia vracia hodnotu typu *unsigned short int*. Jej návratová hodnota je 0, ak je vstupné dvojrozmerné pole, uložené v parametri *this*, neutriedené. Naopak, ak je pole utriedené, funkcia vráti hodnotu 1. Pole je utriedené vtedy, ak sú jeho prvky zoradené v abecednom poradí a teda pre každý prvok poľa platí, že je väčší alebo rovný ako predchádzajúci prvok. Funkcia v cykle porovnáva susedné prvky poľa pomocou funkcie *strcmp*, pričom má dva vstupné parametre a to predchádzajúci a aktuálny prvok poľa. Tieto vstupné parametre porovná. Ak je prvý parameter väčší ako druhý, tak návratová hodnota funkcie je väčšia ako 0. Naopak, ak je prvý parameter menší ako druhý, tak návratová hodnota funkcie je menšia ako 0. V prípade, že je hodnota týchto parametrov rovnaká, tak funkcia vráti hodnotu 0. Táto funkcia sa nachádza v knižnici <*string.h*> a použili sme ju aj pri implementácií triediacich algoritmov. Funkcia overuje či je pole usporiadané. Zavolá sa po usporiadaní triediacimi algoritmami.

```
void Sorter_insertionSort(Sorter *this) {
	for (unsigned int i = 1; i < this->dynamicArray2D.rowCount; i++) {
		char *item_to_sort = this->dynamicArray2D.array[i];
		unsigned int j = i;
		while (j > 0 && strcmp(this->dynamicArray2D.array[j - 1],
item_to_sort) > 0) {
			this->dynamicArray2D.array[j] = this->dynamicArray2D.array[j -
1];
			j--;
		}
		this->dynamicArray2D.array[j] = item_to_sort;
	}
}
```

Obrázok č. 22: Zdrojový kód funkcie *Sorter_insertionSort* [Zdroj: vlastné spracovanie]

Funkcia utriedi pole, ku ktorému pristupuje cez parameter *this*, triediacim algoritmom Insertion Sort. Princíp jeho fungovania sme si priblížili v kapitole 1.6.4. Funkcia nemá návratovú hodnotu, keďže triedenie prebieha priamo v poli. Pole sa prechádza postupne od najmenšieho indexu. Hodnota uložená na tomto indexe sa uloží do pomocnej premennej *item_to_sort*. Aktuálny index *i* sa uloží do premennej *j*. Následne sa vykoná podmienený cyklus, ktorý zabezpečí, že sa aktuálny prvok vloží na správne miesto v poli. Jeho podmienka je, že premenná *j* musí byť väčšia ako 0 a taktiež prvok uložení na indexe *j-1* musí byť väčší ako prvok na indexe *j*. Funkcia porovnáva prvky poľa opäť pomocou funkcie *strcmp*. Pri každej iterácií v tomto cykle sa prvok na mieste *j-1* presunie na miesto *j* a následne sa hodnota premennej *j* zníži o 1. Nakoniec sa uložený prvok v premennej *item_to_sort* vloží do poľa na index *j*.

```
void Sorter_heapify(Sorter *this, unsigned int parentIndex, unsigned int heapLength)
{
	while (1) {
		unsigned int largestIndex = parentIndex;
		unsigned int leftIndex = 2 * parentIndex + 1;
		unsigned int rightIndex = 2 * parentIndex + 2;
		if (leftIndex < heapLength && strcmp(this-
>dynamicArray2D.array[leftIndex], this->dynamicArray2D.array[largestIndex]) > 0) {
			largestIndex = leftIndex;
		}
		if (rightIndex < heapLength && strcmp(this-
>dynamicArray2D.array[rightIndex], this->dynamicArray2D.array[largestIndex]) > 0) {
			largestIndex = rightIndex;
		}
		if (largestIndex != parentIndex) {
			char *parentArray = this->dynamicArray2D.array[parentIndex];
			this->dynamicArray2D.array[parentIndex] = this-
>dynamicArray2D.array[largestIndex];
			this->dynamicArray2D.array[largestIndex] = parentArray;
			parentIndex = largestIndex;
		}
		else {
			return;
		}
	}
}
```

Obrázok č. 23: Zdrojový kód funkcie *Sorter_heapify* [Zdroj: vlastné spracovanie]

Táto funkcia je súčasťou triediaceho algoritmu Heap Sort. Slúži na preusporiadanie prvkov v poli tak, aby spĺňalo podmienky haldy. To znamená, že každý prvok musí byť menší alebo rovný ako jeho rodič. Ako sme si už priblížili bližšie v kapitole 1.6.5.1, prvky haldy vieme dostať nasledovne:

- ľavý potomok = 2 * *aktuálny_index* + 1
- pravý potomok = 2 * *aktuálny_index* + 2
- rodič = (*aktuálny_index* – 1) / 2

Oproti pseudokódu, ktorý sme si ukázali v kapitole 1.6.5.2, si môžeme všimnúť jednu zmenu a to takú, že túto funkciu sme upravili tak, aby nebola rekurzívna. Rekurzívne funkcie sú v programovacom jazyku C pomalšie a teda prepísaním na iteráciu sme dostali efektívnejší kód. Zrýchlilo nám to triediaci algoritmus Heap Sort o približne 50% oproti rekurzívnej funkcie.

```
void Sorter_heapSort(Sorter *this) {
	for (unsigned int parentIndex = this->dynamicArray2D.rowCount / 2;
parentIndex > 0; parentIndex--) {
		Sorter_heapify(this, parentIndex - 1, this->dynamicArray2D.rowCount);
	}
	for (unsigned int remaining = this->dynamicArray2D.rowCount; remaining > 0;
remaining--) {
		char *key = this->dynamicArray2D.array[remaining - 1];
		this->dynamicArray2D.array[remaining - 1] = this-
>dynamicArray2D.array[0];
		this->dynamicArray2D.array[0] = key;
		Sorter_heapify(this, 0, remaining - 1);
	}
}
```

Obrázok č. 24: Zdrojový kód funkcie *Sorter_heapSort* [Zdroj: vlastné spracovanie]

Funkcia vykoná triedenie na vstupnom poli triediacim algoritmom Heap Sort. Najprv sa pole preusporiada tak, aby spĺňalo podmienku haldy. To sa deje postupným volaním funkcie *Sorter_heapify*. Následne ak pole spĺňa podmienku haldy, tak sa vymení prvý a posledný prvok poľa a pole sa opäť preusporiada, aby spĺňalo podmienku haldy. Prvky, ktoré sme vymenili a nachádzajú sa na konci poľa sú už usporiadané a nebudú sa už preusporiadávať na haldu. Po utriedení posledného prvku z haldy funkcia skončí a pole je utriedené týmto triediacim algoritmom. Taktiež keďže triediaci algoritmus vykonáva triedenie na vstupnom poli, nemá žiadnu návratovú hodnotu.

```
void Sorter_timeSort(Sorter *this, unsigned int sortType) {
	if (sortType == 1) {
		printf("\nVstupny subor sa utrieduje algoritmom Insertion Sort.\n");
		this->timeTracker.startTime = clock();
		Sorter_insertionSort(this);
		this->timeTracker.endTime = clock();
		TimeTracker_fillDuration(&this->timeTracker);
		printf("Cas triedenia algoritmom Insertion Sort: %fs\n", this-
>timeTracker.duration);
	}
	else if (sortType == 2) {
		printf("\nVstupny subor sa utrieduje algoritmom Heap Sort.\n");
		this->timeTracker.startTime = clock();
		Sorter_heapSort(this);
		this->timeTracker.endTime = clock();
		TimeTracker_fillDuration(&this->timeTracker);
		printf("Cas triedenia algoritmom Heap Sort: %fs\n", this-
>timeTracker.duration);
	}
	printf("Je subor utriedeny? %s\n", Sorter_isSorted(this) ? "Ano" : "Nie");
}
```

Obrázok č. 25: Zdrojový kód funkcie *Sorter_timeSort* [Zdroj: vlastné spracovanie]

V tejto funkcií sa zaznamenáva trvanie triediaceho algoritmu. Aký triediaci algoritmus sa má vykonať, záleží od vstupného parametra *sortType*. Ak sa rovná 1, tak sa vykoná a odmeria čas triediaceho algoritmu Insertion Sort. Ak sa rovná 2, tak sa vykoná a odmeria triediaci algoritmus Heap Sort. Najprv sa teda vypíše informácia na konzolu, aký triediaci algoritmus sa ide spustiť na vstupnom poli. Následne sa triediaci algoritmus vykoná a jeho čas sa odmeria. Nakoniec sa vypíše informácia o tom ako dlho trvalo triedenie a následne či je pole utriedené.

```
void Sorter_saveToLogFile(Sorter *this, char inputFileName[], char sortedFileName[],
unsigned int sortType) {
	struct tm tm;
	time_t now = time(0);
	localtime_s(&tm, &now);
	FILE *file = fopen(LOG_FILENAME, "a");
	fprintf(file, "Datum a cas: %02d.%02d.%d %02d:%02d:%02d\n", tm.tm_mday,
tm.tm_mon, 1900 + tm.tm_year, tm.tm_hour, tm.tm_min, tm.tm_sec);
	fprintf(file, "Nazov vstupneho suboru s datami, ktore program usporiada:
%s\n", inputFileName);
	fprintf(file, "Nazov vystupneho suboru s usporiadanymi datami: %s\n",
sortedFileName);
	fprintf(file, "Pocet riadkov: %d\n", this->dynamicArray2D.rowCount);
	fprintf(file, "Pocet znakov najdlhsieho riadku: %d\n", this-
>dynamicArray2D.colCount);
	if (sortType == 1) {
		fprintf(file, "Nazov triediaceho algoritmu: Insertion Sort\n");
	}
	else if (sortType == 2) {
		fprintf(file, "Nazov triediaceho algoritmu: Heap Sort\n");
	}
	fprintf(file, "Cas: %fs\n\n", this->timeTracker.duration);
	fclose(file);
}
```

Obrázok č. 26: Zdrojový kód funkcie *Sorter_saveToLogFile* [Zdroj: vlastné spracovanie]

Táto funkcia ukladá údaje o vykonanom triedení do logovacieho súboru. Logovací súbor sa otvorí v móde „a“, čo znamená, že sa bude zapisovať na koniec súboru. Prvým riadkom tohto súboru je dátum a čas zápisu tohto logu. Nasleduje názov vstupného súboru, ktorý program načítal a utriedil. Ďalším riadkom je názov výstupného utriedeného súboru. Tieto dva riadky slúžia na spárovanie týchto súborov, aby sme vedeli zistiť, ktorý neutriedený súbor patrí ku ktorému utriedenému. Vo štvrtom riadku logu sa nachádza počet riadkov triedeného súboru a v piatom počet znakov najdlhšieho riadku. Pomocou týchto riadkov vieme spätne zistiť aké veľké dynamické pole program alokoval. To zároveň aj ovplyvnilo čas potrebný na utriedenie polí. Čím väčšie budú tieto hodnoty, tým dlhšie bude trvať utriedenie súboru. Ďalej je súčasťou logu riadok vyjadrujúci názov triediaceho algoritmu. Ten sa zistí, ako aj v predchádzajúcej funkcii, z parametra *sortType*. Posledný riadok je najdôležitejší a to dĺžka trvania triedenia v sekundách.

```
void executeSorts(char inputFileName[]) {
	Utils_createHeaderCSV();

	DynamicArray1D dynamicArray1D;
	DynamicArray1D_createFromFile(&dynamicArray1D, inputFileName);

	DynamicArray2D dynamicArray2D;
	DynamicArray2D_setLengthsFromString(&dynamicArray2D, &dynamicArray1D, '\n');

	printf("\nPocet riadkov suboru, ktory sa utriedi: %d.\n",
dynamicArray2D.rowCount);
	double ram = dynamicArray2D.colCount * dynamicArray2D.rowCount +
dynamicArray2D.rowCount * sizeof(char *) + dynamicArray1D.count;
	printf("\nProgram bude potrebovat priblizne aspon %fMB volnej RAM pamate.",
ram / 1048576);
	unsigned int continueProgram = Utils_getInputBool("\nMa program pokracovat a
utriedit subor? (A/N): ");
	if (!continueProgram) {
		free(dynamicArray1D.array);
		return;
	}

	FILE *fileCSV = fopen(CSV_FILENAME, "a");
	fprintf(fileCSV, "\n%d;%d", dynamicArray2D.rowCount,
dynamicArray2D.colCount);
	fclose(fileCSV);

	Sorter sorter;
	char outputFileName[100] = "";
	for (unsigned int i = 1; i <= 2; i++) {
		DynamicArray2D_fillFromString(&dynamicArray2D, &dynamicArray1D, '\n');
		sorter.dynamicArray2D = dynamicArray2D;
		Sorter_timeSort(&sorter, i);
		DynamicArray2D_saveToFile(&sorter.dynamicArray2D, &outputFileName);
		printf("Utriedeny subor '%s' bol zapisany.\n", outputFileName);
		Sorter_saveToLogFile(&sorter, inputFileName, outputFileName, i);
		DynamicArray2D_free(&sorter.dynamicArray2D);

		fileCSV = fopen(CSV_FILENAME, "a");
		fprintf(fileCSV, ";%f", sorter.timeTracker.duration);
		fclose(fileCSV);
	}

	free(dynamicArray1D.array);
}
```

Obrázok č. 27: Zdrojový kód funkcie *executeSorts* [Zdroj: vlastné spracovanie]

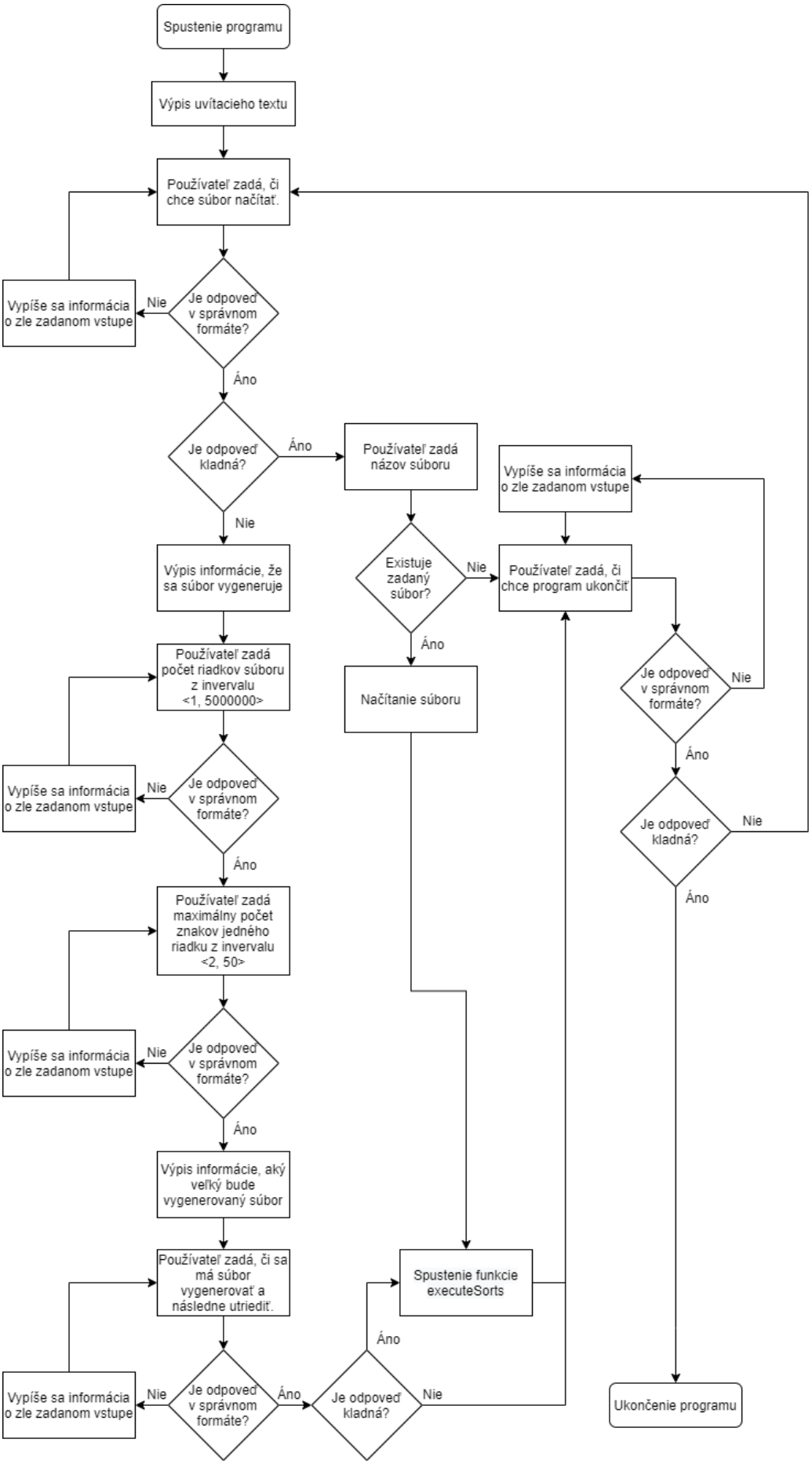
Funkcia zavolá funkciu na pripravenie triedenia a funkciu na utriedenie poľa na vstupnom súbore. Následne zapíše výsledok do CSV dokumentu ako aj zavolá funkciu na zapisovanie logovacieho súboru a funkciu na zapísanie utriedeného súboru. Pred načítaním dvojrozmerného dynamického poľa z jednorozmerného sa program spýta používateľa či chce pokračovať. Zobrazí sa informácia o potrebnej voľnej veľkosti na RAM pamäti v megabajtoch na vytvorenie tohto dvojrozmerného poľa k už využitému miestu jednorozmerného poľa. Triedenie prebieha v cykle, pričom sa najprv vykoná Insertion Sort a následne Heap Sort. Po utriedení jednotlivými algoritmami funkcia uvoľní z RAM pamäte dvojrozmerné pole a opäť sa načíta z jednorozmerného. Obsah súboru, ktorý sa má utriediť

si funkcia uchováva v jednorozmernom poli, aby sa zbytočne súbor neotváral viackrát, čím by sa algoritmus spomalil a čím sme zaručili, že sa utrieďuje ten istý obsah. Keďže sa obsah súboru utrieďuje v dvojrozmernom poli, ktoré síce vytvárame z jednorozmerného, tak sa toto jednorozmerné pole neutrieďuje. Na druhej strane, vďaka tomu pole zaberá viacej miesta v RAM pamäti.

```
void main();
```

Obrázok č. 28: Zdrojový kód hlavičky funkcie *main* [Zdroj: vlastné spracovanie]

Pri spustení programu sa táto funkcia zavolá ako prvá. Nachádzajú sa v nej najmä príkazy na vypísanie textu a príkazy na spracovanie textu od používateľa. Keďže má funkcia väčší obsah, tak si ukážeme jej funkčnosť na vývojovom diagrame.



Obrázok č. 29: Vývojový diagram funkcie *main* [Zdroj: vlastné spracovanie]

Ako je možné vidieť na vývojovom diagrame, funkcia slúži najmä na spracovanie vstupu od používateľa. Používateľ vie zadať súbor, ktorý sa má načítať a následne sa na ňom spustí funkcia *executeSorts*, ktorá usporiada súbor dvomi triediacimi algoritmami. V prípade, že používateľ nechce načítať súbor, sa súbor vygeneruje podľa parametrov zadaných používateľom, tento súbor sa zapíše na disk a následne sa na ňom zavolá funkcia *executeSorts*. Jednotlivé vstupy zadané používateľom sú ošetrené a pri zle zadanom formáte sa vypíše chybová hláška. Na to, aby sa používateľ dostal na ďalší krok, musí zadať odpoveď v správnom formáte. Pri generovaní súboru sa zároveň vypočíta a zobrazí používateľovi informácia o možnej veľkosti súboru a predtým ako sa vytvorí a zapíše to musí odsúhlasiť používateľ.

4.2 Funkčnosť programu

Vytvorili sme program, ktorý dokáže zmerať exekučnú efektívnosť triediacich algoritmov Heap Sort a Insertion Sort. Tieto časy zobrazí aj do konzoly a aj ich uloží do dvoch súborov.

```
Nacitat zo subora? (A/N): n
Subor, ktory sa ma usporiadat sa vygeneruje z kombinacii malych znakov abecedy.

Zadajte cislo vyradrujuce pocet riadkov suboru z intervalu <1, 5000000>: 1000
Zadajte cislo vyjadrujuce maximalny pocet znakov jedneho riadku suboru z intervalu <2,
100>: 10

Program vygeneruje subor, ktoreho velkost je aspon 0.001907MB a najviac 0.010490MB.
Taktiez na jeho utriedenie bude pozadovana aspon takato volna velkost RAM pamate.
Ma program pokracovat a vytvorit subor, ktory sa ma utriedit? (A/N): a

Subor 'vstupne subory/2021-04-14 10-08-59 9507.txt' bol vytvoreny.
Pocet riadkov suboru, ktory sa utriedi: 1000.

Program bude potrebovat priblizne aspon 0.021396MB volnej RAM pamate.
Ma program pokracovat a utriedit subor? (A/N): a

Vstupny subor sa utrieduje algoritmom Insertion Sort.
Cas triedenia algoritmom Insertion Sort: 0.001000s
Je subor utriedeny? Ano
Utriedeny subor 'usporiadane subory/2021-04-14 10-09-02 5389.txt' bol zapisany.

Vstupny subor sa utrieduje algoritmom Heap Sort.
Cas triedenia algoritmom Heap Sort: 0.000000s
Je subor utriedeny? Ano
Utriedeny subor 'usporiadane subory/2021-04-14 10-09-02 6960.txt' bol zapisany.
-------------------------
Chcete program ukoncit? (A/N): a
```

Obrázok č. 30: Ukážka výpisu programu do konzoly [Zdroj: vlastné spracovanie]

Vo výpise programu je vidieť ako program najprv požaduje od používateľa jednotlivé parametre. Pri generácií súboru sa vypíše aj možná veľkosť súboru. Následne program aj zobrazí informáciu o požadovanej voľnej veľkosti na RAM pamäti. Po tom, čo program načíta obsah súboru do pamäte ho následne usporiada oboma algoritmami a zapíše do súboru. Do konzoly sa vypíše informácia o čase, ktorý program potreboval na utriedenie obsahu vstupného súboru ako aj to, kde sa výsledný súbor zapísal. Taktiež sa spustí kontrola, pred zápisom do súboru, či je súbor naozaj usporiadaný.

Informácia o spustení programu sa uchovala aj do logovacieho súboru *log.txt*, kde sa po utriedení ukladali dôležité parametre o danom triedení.

```
Datum a cas usporiadania:                                   14.04.2021 10:09:02
Nazov vstupneho suboru s datami, ktore program usporiadal:  vstupne subory/2021-
04-14 10-08-59 9507.txt
Nazov vystupneho suboru s usporiadanymi datami:             usporiadane
subory/2021-04-14 10-09-02 5389.txt
Pocet riadkov:                                              1000
Pocet znakov najdlhsieho riadku:                            10
Nazov triediaceho algoritmu:                                Insertion Sort
Cas:                                                        0.001000s

Datum a cas usporiadania:                                   14.04.2021 10:09:02
Nazov vstupneho suboru s datami, ktore program usporiadal:  vstupne subory/2021-
04-14 10-08-59 9507.txt
Nazov vystupneho suboru s usporiadanymi datami:             usporiadane
subory/2021-04-14 10-09-02 6960.txt
Pocet riadkov:                                              1000
Pocet znakov najdlhsieho riadku:                            10
Nazov triediaceho algoritmu:                                Heap Sort
Cas:                                                        0.000000s
```

Obrázok č. 31: Obsah logovacieho súboru [Zdroj: vlastné spracovanie]

Na obrázku č. 31 môžeme vidieť jednotlivé parametre usporiadania. Do logovacieho súboru sa zapisujú dôležité informácie, a to najmä názov vstupného a výstupného súboru. Taktiež dátum usporiadania a čas jednotlivých triedení. Výsledok z konzoly sme teda nemuseli manuálne ukladať, ale program to spravil za nás.

Parametre triedení sme si taktiež zapisovali do súboru *porovnanie.csv*. Tento súbor nám slúžil na uľahčenie pri vytváraní tabuliek a grafov v nasledujúcej kapitole. Zapisujú sa tu informácie o počte riadkov a počte najdlhšieho riadku, na ktorom sa triedenie vykonalo a aj časy jednotlivých triediacich algoritmov.

```
pocet riadkov;najvacsi pocet znakov jedneho riadku;Insertion Sort (s);Heap Sort (s)
1000;10;0.001000;0.000000
```

Obrázok č. 32: Obsah súboru *porovnanie.csv* [Zdroj: vlastné spracovanie]

4.3 Výsledky meraní exekučných časov

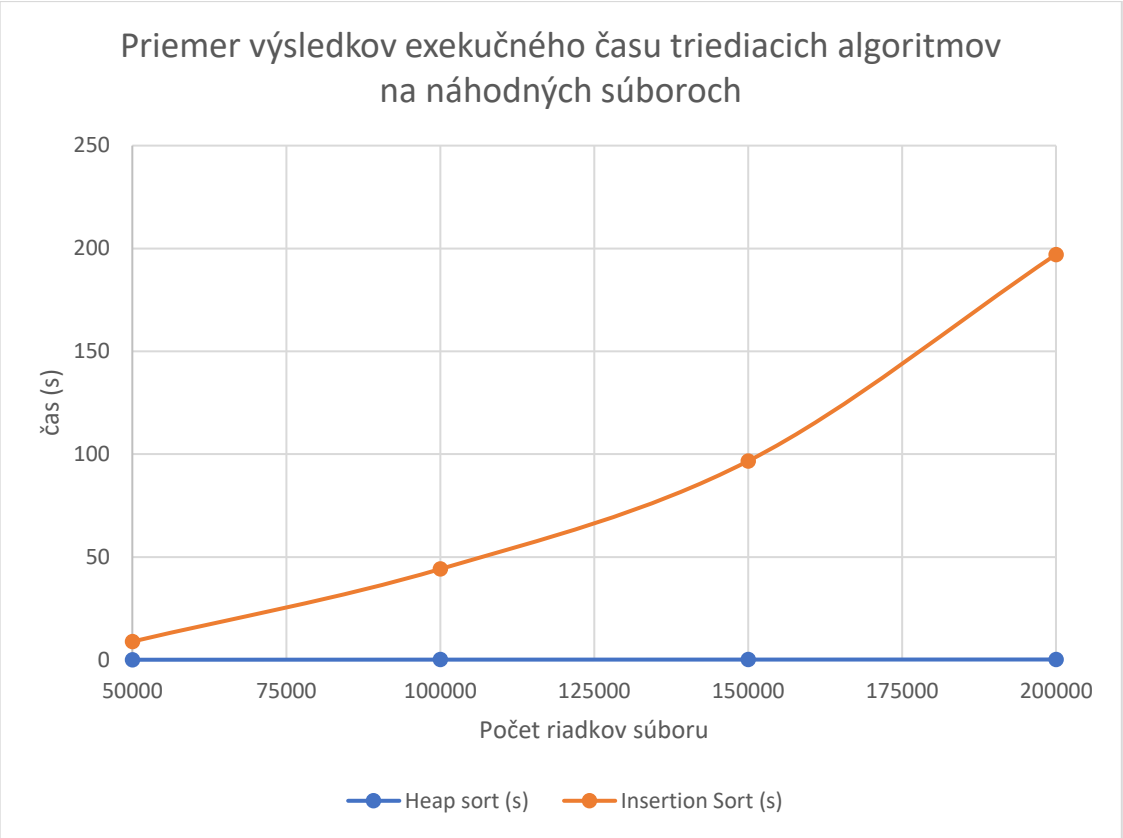
4.3.1 Výsledky meraní exekučných časov na náhodných súboroch

Najprv sme pomocou programu vygenerovali vstupné súbory a tie sme následne utrieďovali triediacimi algoritmami Insertion Sort a Heap Sort. Súbory obsahovali náhodné znaky malej latinskej abecedy. Vstupných súborov sme vytvorili štyridsať, pričom boli rozdelené do štyroch skupín po desať, podľa nasledovných počtov riadkov: 50 000, 100 000, 150 000 a 200 000. Maximálny počet znakov jedného riadku bol 50. Následne sme vypočítali priemer trvania usporiadania triedenia podľa počtu riadkov súborov.

Počet riadkov	**Insertion Sort (s)**	**Heap Sort (s)**
50000	8.878	0.0507
100000	44.1484	0.1118
150000	96.6423	0.1285
200000	197.065	0.1753

Tabuľka č. 1: Spriemerované výsledky meraní exekučných časov triediacich algoritmov Insertion Sort a Heap Sort na náhodných súboroch [Zdroj: vlastné spracovanie]

Z nameraných výsledkov môžeme vidieť, že Insertion Sort je jasne pomalší. Ako sme si už spomenuli v kapitole 1.6.4.2, priemerná časová zložitosť tohto algoritmu je $O(n^2)$. To znamená, že ak sa počet riadkov súboru zdvojnásobí, čas by mal narásť štvornásobne. Ak porovnáme prvý a druhý riadok ako aj druhý a štvrtý, tak sa nám namerané časy zvýšili približne o štvornásobok. Z tabuľky sme vytvorili graf na ktorom je vidieť, že Insertion Sort je jasne pomalší a neefektívny.



Graf č. 2: Priemer výsledkov exekučného času triediacich algoritmov na náhodných súboroch [Zdroj: vlastné spracovanie]

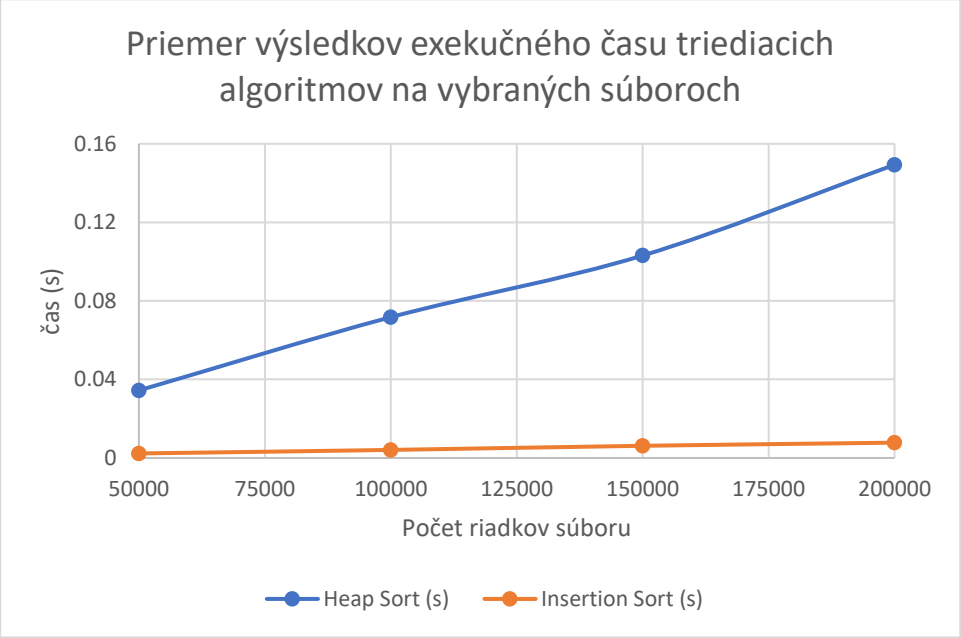
4.3.2 Výsledky meraní exekučných časov na vybraných súboroch

Toto meranie sme uskutočnili na súboroch, ktoré boli takmer usporiadané. Na to aby boli usporiadané, bolo potrebné vymeniť dva susedné prvky. Vstupné súbory, na ktorých sme uskutočnili toto meranie, sme dostali pri predchádzajúcom meraní. Vybrali sme jeden súbor z každej skupiny, pričom tieto skupiny boli také isté ako pri predchádzajúcom meraní a to: 50 000, 100 000, 150 000 a 200 000. Následne sme v nich vymenili dva susedné prvky tak, aby súbor nebol utriedený. Na upravených súboroch sme spustili desaťkrát náš program.

Počet riadkov	**Insertion Sort (s)**	**Heap Sort (s)**
50000	0.0023	0.0344
100000	0.0041	0.0717
150000	0.0062	0.1031
200000	0.0078	0.1493

Tabuľka č. 2: Spriemerované výsledky meraní exekučných časov triediacich algoritmov Insertion Sort a Heap Sort na vybraných súboroch [Zdroj: vlastné spracovanie]

Z nameraných výsledkov si môžeme všimnúť, že Insertion Sort bol v tomto meraní rýchlejší ako Heap Sort. Je to kvôli tomu, že Insertion Sort sa utriedťuje na súbore, ktorý má vymenené iba dva susedné prvky. Insertion Sort vie tieto prvky ľahko vymeniť vďaka čomu súbor usporiada rýchlejšie ako Heap Sort. Insertion Sort teda v tomto prípade nemá časovú zložitosť $O(n^2)$, ale v podstate $O(n)$. Z nameraných výsledkov sme vytvorili graf.



Graf č. 3: Priemer výsledkov exekučného času triediacich algoritmov na vybraných súboroch [Zdroj: vlastné spracovanie]

4.3.3 Záver meraní

Z nameraných výsledkov sme dospeli k záveru, že Heap Sort je jednoznačne rýchlejší na súbore, ktorého obsah je náhodne vygenerovaný a je teda potrebné na jeho utriedenie presunúť väčšinu prvkov na správne miesto. Insertion Sort je v tomto prípade neefektívny a trvá oveľa dlhšie. Pri 200 000 riadkoch bol Insertion Sort až približne tisíckrát pomalší, čím sa stáva neefektívny na triedenie veľkých súborov s neusporiadanými dátami.

Naopak z výsledkov druhého merania sme dospeli k záveru, že ak poznáme obsah súboru a stačí nám vymeniť pár prvkov, tak je Insertion Sort rýchlejší. Síce sme usporadúvali súbory, ktoré sú už takmer utriedené, ale práve kvôli tomu bolo triedenie algoritmom Insertion Sort rýchlejšie. Heap Sort má vždy rovnakú časovú zložitosť a práve preto v tomto meraní zaostával. Insertion Sort bol pri 200 000 riadkových súboroch približne dvadsaťkrát rýchlejší.

Záver

Výsledkom práce bola analýza triediacich algoritmov s bližším zameraním na Heap Sort a Insertion Sort. Cieľom bolo vytvoriť program, ktorý porovná exekučnú efektívnosť týchto triediacich algoritmov. Nami vytvorený program dokázal uskutočniť tieto porovnania a zároveň nám pomohol pri vytvorení súborov, ktoré sa mali usporiadať. Program pracuje na základe poskytnutých parametrov od používateľa, vďaka čomu je možné ho použiť aj na inom zariadení a na iných súboroch. Výsledok programu sa uchováva v logovacom súbore, vďaka čomu je možné si kedykoľvek pozrieť prehľad jednotlivých usporiadaní.

Na základe výsledkov meraní, ktoré sme dostali pomocou programu, sme dospeli k záveru, že exekučná efektívnosť triediaceho algoritmu Heap Sort na neusporiadaných súboroch bola niekoľkonásobne rýchlejšia oproti triediacemu algoritmu Insertion Sort. Naopak na takmer utriedených súboroch bol rýchlejší Insertion Sort. Preto by sa oba triediace algoritmy mali používať v závislosti na obsahu súboru, ktorý chceme utriediť.

Zoznam použitej literatúry

[1] Algorithms in C. by Robert Sedgewick, Addison-Wesley, 1990, ISBN 0-201-51425-7

[2] http://input.sk/python2017/31.html (1. máj 2021)

[3] http://input.sk/python2016/12.html (1. máj 2021)

[4] https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/ (1. máj 2021)

[5] https://www.geeksforgeeks.org/stability-in-sorting-algorithms/ (1. máj 2021)

[6] https://adrianmejia.com/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/ (1. máj 2021)

[7] https://www.geeksforgeeks.org/bubble-sort/ (1. máj 2021)

[8] https://www.geeksforgeeks.org/selection-sort/ (1. máj 2021)

[9] https://www.geeksforgeeks.org/quick-sort/ (1. máj 2021)

[10] https://www.geeksforgeeks.org/insertion-sort/ (1. máj 2021)

[11] https://www.geeksforgeeks.org/heap-sort/ (1. máj 2021)

[12] http://input.sk/struct2017/04.html (1. máj 2021)

Zoznam ilustrácií a zoznam tabuliek a grafov

Zoznam obrázkov

Obrázok č. 1: Pseudokód triediaceho algoritmu Bubble Sort [Zdroj: vlastné spracovanie]

Obrázok č. 2: Pseudokód triediaceho algoritmu Selection Sort [Zdroj: vlastné spracovanie]

Obrázok č. 3: Pseudokód triediaceho algoritmu Quick Sort [Zdroj: vlastné spracovanie]

Obrázok č. 4: Pseudokód triediaceho algoritmu Insertion Sort [Zdroj: vlastné spracovanie]

Obrázok č. 5: Pseudokód triediaceho algoritmu Heap Sort [Zdroj: vlastné spracovanie]

Obrázok č. 6: Mapa volaní funkcií [Zdroj: vlastné spracovanie]

Obrázok č. 7: Diagram štruktúr [Zdroj: vlastné spracovanie]

Obrázok č. 8: Zdrojový kód funkcie *Utils_getRandomNumberWithEqualProbability* [Zdroj: vlastné spracovanie]

Obrázok č. 9: Zdrojový kód funkcie *Utils_generateFileName* [Zdroj: vlastné spracovanie]

Obrázok č. 10: Zdrojový kód funkcie *Utils_generateNonExistingFileName* [Zdroj: vlastné spracovanie]

Obrázok č. 11: Zdrojový kód funkcie U*tils_generateFileWithRandomContentOfSmallAlphabet* [Zdroj: vlastné spracovanie]

Obrázok č. 12: Zdrojový kód hlavičiek ostatných funkcií v súbore *Utils.h* [Zdroj: vlastné spracovanie]

Obrázok č. 13: Zdrojový kód funkcie *TimeTracker_fillDuration* [Zdroj: vlastné spracovanie]

Obrázok č. 14: Zdrojový kód funkcie *DynamicArray1D_createEmpty* [Zdroj: vlastné spracovanie]

Obrázok č. 15: Zdrojový kód funkcie *DynamicArray1D_createFromFile* [Zdroj: vlastné spracovanie]

Obrázok č. 16: Zdrojový kód funkcie *DynamicArray2D_createEmpty* [Zdroj: vlastné spracovanie]

Obrázok č. 17: Zdrojový kód funkcie *DynamicArray2D_free* [Zdroj: vlastné spracovanie]

Obrázok č. 18: Zdrojový kód funkcie *DynamicArray2D_setLengthsFromString* [Zdroj: vlastné spracovanie]

Obrázok č. 19: Zdrojový kód funkcie *DynamicArray2D_fillFromString* [Zdroj: vlastné spracovanie]

Obrázok č. 20: Zdrojový kód funkcie *DynamicArray2D_saveToFile* [Zdroj: vlastné spracovanie]

Obrázok č. 21: Zdrojový kód funkcie *Sorter_isSorted* [Zdroj: vlastné spracovanie]

Obrázok č. 22: Zdrojový kód funkcie *Sorter_insertionSort* [Zdroj: vlastné spracovanie]

Obrázok č. 23: Zdrojový kód funkcie *Sorter_heapify* [Zdroj: vlastné spracovanie]

Obrázok č. 24: Zdrojový kód funkcie *Sorter_heapSort* [Zdroj: vlastné spracovanie]

Obrázok č. 25: Zdrojový kód funkcie *Sorter_timeSort* [Zdroj: vlastné spracovanie]

Obrázok č. 26: Zdrojový kód funkcie *Sorter_saveToLogFile* [Zdroj: vlastné spracovanie]

Obrázok č. 27: Zdrojový kód funkcie *executeSorts* [Zdroj: vlastné spracovanie]

Obrázok č. 28: Zdrojový kód hlavičky funkcie *main* [Zdroj: vlastné spracovanie]

Obrázok č. 29: Vývojový diagram funkcie *main* [Zdroj: vlastné spracovanie]

Obrázok č. 30: Ukážka výpisu programu do konzoly [Zdroj: vlastné spracovanie]

Obrázok č. 31: Obsah logovacieho súboru [Zdroj: vlastné spracovanie]

Obrázok č. 32: Obsah súboru *porovnanie.csv* [Zdroj: vlastné spracovanie]

Zoznam tabuliek

Tabuľka č. 1: Spriemerované výsledky meraní exekučných časov triediacich algoritmov Insertion Sort a Heap Sort na náhodných súboroch [Zdroj: vlastné spracovanie]

Tabuľka č. 2: Spriemerované výsledky meraní exekučných časov triediacich algoritmov Insertion Sort a Heap Sort na vybraných súboroch [Zdroj: vlastné spracovanie]

Zoznam grafov

Graf č. 1: Grafické znázornenie výpočtovej zložitosti [Zdroj: vlastné spracovanie]

Graf č. 2: Priemer výsledkov exekučného času triediacich algoritmov na náhodných súboroch [Zdroj: vlastné spracovanie]

Graf č. 3: Priemer výsledkov exekučného času triediacich algoritmov na vybraných súboroch [Zdroj: vlastné spracovanie]

Prílohy

Zdrojový kód programu, samotný program a textové súbory, ktoré vznikli pri meraní

Manuál k programu