EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

POROVNANIE EXEKUČNEJ EFEKTÍVNOSTI TRIEDIACICH ALGORITMOV SHAKE SORT A SHELL SORT USPORADÚVAJÚCICH VEĽKÉ SÚBORY DÁT V PROGRAME VYTVORENOM V JAZYKU C

Bakalárska práca

2020 **Lukáš Šury**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

POROVNANIE EXEKUČNEJ EFEKTÍVNOSTI TRIEDIACICH ALGORITMOV SHAKE SORT A SHELL 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.

2020 **Lukáš Šury**

Čestné vyhlásenie

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

V Trnave, 16. 05. 2020 .....

Podpis

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Lukáš Šury
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 Shake Sort a Shell 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 Shake Sort a Shell Sort. V rámci bakalárskej práce študent vytvorí program v jazyku C, v ktorom implementuje oba triediace algoritmy Shake Sort a Shell 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 Shake Sort a Shell 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: 19.04.2017

Ing. Mgr. Peter Schmidt, PhD.
vedúci katedry

ABSTRAKT

ŠURY, Lukáš: *Porovnanie exekučnej efektívnosti triediacich algoritmov Shake sort a Shell 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, 2020, 47 s.

Cieľom bakalárskej práce je opísať pojem algoritmus a zanalyzovať vybrané druhy triediacich algoritmov. Následne vytvoriť program v jazyku C s implementovanými triediacimi algoritmami Shell sort a Shake sort, ktoré budú usporadúvať veľké celočíselné polia. Exekučné časy týchto usporadúvaní budeme porovnávať. V prvej kapitole sa zaoberáme definíciou pojmu algoritmus. V druhej kapitole sa zaoberáme analýzou algoritmov, ich klasifikáciou a ich výpočtovou zložitosťou. Tretia kapitola sa venuje vybraným druhom triediacich algoritmov. Štvrtá kapitola opisuje vytvorený program v jazyku C a implementáciou triediacich algoritmov Shake sort a Shell sort. Piata kapitola je zameraná na meranie exekučných časov týchto dvoch algoritmov a ich porovnávanie.

Kľúčové slová: algoritmus, triedenie, Shake sort, Shell sort, jazyk C

ABSTRACT

ŠURY, Lukáš: *A comparison of an execution efficiency of Shake Sort and Shell Sort sorting algorithms that sort large data sets in a program created in C* – University of economics in Bratislava. Faculty of economic informatics; Department of applied informatics. – Thesis supervisor: Ing. Igor Košťál, PhD. – Bratislava: FEI, 2020, 47 p.

The aim of the bachelor thesis is to describe the term algorithm and analyze selected types of sorting algorithms. Then create a program in C language with implemented sorting algorithms Shell sort and Shake sort, which will sort large integer arrays. We will compare the execution times of these sorts. In the first chapter we deal with the definition of the term algorithm. In the second chapter we deal with the analysis of algorithms, their classification and their computational complexity. The third chapter deals with selected types of sorting algorithms. The fourth chapter describes the created program in C language and the implementation of sorting algorithms Shake sort and Shell sort. The fifth chapter is focused on measuring the execution times of these two algorithms and comparing them.

Keywords: algorithm, sorting, Shake sort, Shell sort, C language

Obsah

Zoznam ilustrácií

Zoznam tabuliek

Úvod

Triedenie je základným stavebným blokom, okolo ktorého je zostavených mnoho ďalších algoritmov. Pochopením triedenia získame veľké množstvo možností na vyriešenie ďalších problémov.

V minulosti počítače spotrebúvali viac času triedením ako čímkoľvek iným. Aj keď nie je jasné, či to platí aj pre dnešné počítače, triedenie zostáva v praxi najbežnejším problémom algoritmov kombinatoriky.

Triedenie je jeden z najdôkladnejšie rozoberaných problémov počítačovej vedy. Doslova sú známe desiatky rôznych algoritmov, z ktorých väčšina má v určitých situáciách určitú výhodu oproti všetkým ostatným algoritmom. (Skiena, 1997, s. 103)

V tejto práci sa zaoberáme analýzou algoritmov, ich časovou zložitosťou a následnou analýzou vybraných druhov triediacich algoritmov. Ďalej pomocou vývojového prostredia Visual Studio 2019 od firmy Microsoft vytvoríme program napísaný v programovacom jazyku C. Opíšeme jeho základné funkcie a výstupy. Do tohto programu implementujeme triediace algoritmy Shake sort a Shell sort. Otestujeme a porovnáme ich exekučnú efektívnosť pri usporadúvaní veľkého množstva dát, napríklad celočíselného poľa so stotisíc prvkami.

1 Úloha algoritmov v oblasti výpočtovej techniky

Pri písaní počítačového programu sa všeobecne zavádza taká metóda riešenia problému, ktorá bol navrhnutá predtým. Táto metóda je často nezávislá od konkrétneho počítača, ktorý sa má použiť. Je pravdepodobné, že bude rovnako vhodná pre väčšinu počítačov. V každom prípade je potrebné študovať metódu, nie samotný počítačový program, aby sme sa naučili, akým spôsobom je problém riešený. Termín algoritmus sa používa v počítačovej vede na opísanie metódy riešenia problémov, ktorá je vhodná na zavedenie do počítačových programov. Algoritmy sú ústrednými predmetmi štúdia v mnohých oblastiach odboru. Ak sa má vyvinúť veľmi veľký počítačový program, musí sa vynaložiť veľké úsilie na pochopenie a definovanie problému, ktorý sa má vyriešiť a na jeho rozloženie na menšie čiastkové úlohy, ktoré sa dajú ľahko implementovať. (Sedgewick, 1990, s. 4)

Neformálne je algoritmus akýkoľvek dobre definovaný výpočtový postup, ktorý berie ako vstup nejakú hodnotu alebo množinu hodnôt a ako výstup vytvorí nejakú hodnotu alebo množinu hodnôt. Algoritmus je teda postupnosť výpočtových krokov, ktoré transformujú vstup na výstup. Algoritmus môžeme tiež vnímať ako nástroj na riešenie dobre špecifikovaného výpočtového problému. Špecifikovanie problému všeobecne špecifikuje aj požadovaný vzťah medzi vstupom a výstupom. Algoritmus opisuje určitý výpočtový postup na dosiahnutie tohto požadovaného vzťahu medzi vstupom a výstupom. Napríklad by sme mohli potrebovať zoradiť postupnosť čísel do vzostupného poradia. Tento problém sa v praxi často vyskytuje a poskytuje úrodnú pôdu pre zavedenie mnohých štandardných techník navrhovania a analytických nástrojov. Takto formálne definujeme triediaci problém:

Vstup: Sekvencia n čísel ($a_1, a_2, ..., a_n$).
Výstup: Permutácia (zmena poradia) ($a'_1, a'_2, ..., a'_n$) vstupnej sekvencie tak, aby $a'1 \leq a'2 \leq ... \leq a'_n$.

Ak máme napríklad danú vstupnú sekvenciu (31, 41, 59, 26, 41, 58), triediaci algoritmus vracia ako výstup sekvenciu (26, 31, 41, 41, 58, 59). Takáto vstupná sekvencia sa nazýva inštancia triediaceho problému. Všeobecne platí, že inštancia problému pozostáva zo vstupu potrebných na výpočet riešenia problému.

Pretože triedenie mnoho programov používa ako medzistupeň, v počítačovej vede je považované za základnú operáciu. Výsledkom je, že máme k dispozícii veľké množstvo dobrých algoritmov triedenia. Ktorý algoritmus je pre danú aplikáciu najlepší, závisí okrem iného od počtu prvkov, ktoré sa majú triediť, od rozsahu, v akom sú prvky usporiadané pred

začatím triedenia, od možných obmedzení hodnôt položiek, architektúry počítača a druhu úložných zariadení, ktoré sa majú použiť. (Cormen a kol., 2009, s. 5 - 6)

2 Analýza algoritmov

Pre väčšinu problémov je k dispozícii veľa rôznych algoritmov. Ako si teda zvoliť ten správny algoritmus na implementáciu? Toto je v skutočnosti dobre rozvinutá oblasť v počítačovej vede. Existuje veľa výskumov popisujúcich výkon základných algoritmov. Porovnávanie algoritmov však môže byť skutočne náročné a je teda potrebné stanoviť určité všeobecné pokyny. (Sedgewick, 1990, s. 67)

Problémy, ktoré pomocou algoritmov riešime, majú zvyčajne svoju prirodzenú „veľkosť“ (zvyčajne množstvo údajov, ktoré sa majú spracovať), ktorú bežne označujeme N. Použité zdroje (najčastejšie množstvo potrebného času) by sme mohli opísať ako funkciu čísla N. Jednou z vecí, ktorá nás pri analýze algoritmov zaujíma je „priemerný čas“ doby chodu algoritmu, teda množstvo času, ktoré očakávame, že program zaberie pri zadaní „typických“ vstupných údajoch a tiež nás zaujíma „najhorší čas“ doby chodu algoritmu, ktorý by program zabral pri konfigurácii najhorších možných vstupov.
(Sedgewick, 1990, s. 67)

Najhorší čas vykonávania algoritmu nám dáva hornú hranicu doba chodu pre akýkoľvek vstup. Jeho znalosť poskytuje záruku, že algoritmus nikdy nebude trvať dlhšie.
(Cormen a kol., 2009, s. 27)

Pri určovaní priemerného a najhoršieho času trvania algoritmu sú použité presné matematické vzorce. Tieto vzorce sa vyvíjajú starostlivým štúdovaním algoritmu, nájdením doby trvania z hľadiska základných matematických veličín a následnou matematickou analýzou zúčastnených veličín. (Sedgewick, 1990, s. 67)

Do tejto analýzy vstupuje niekoľko dôležitých faktorov, ktoré sú zvyčajne mimo vplyvu programátora. Po prvé, programy vytvorené v jazyku C sú prekladané do strojového kódu pre daný počítač a môže byť náročnou úlohou zistiť, ako dlho môže trvať vykonanie jedného príkazu v jazyku C (najmä v prostredí, kde sa zdieľajú zdroje, takže dokonca aj rovnaký program môže mať rôzne výkonnostné charakteristiky). Po druhé, mnoho programov je mimoriadne citlivých na ich vstupné údaje a výkon sa môže v závislosti od vstupov veľmi meniť. Priemerné časy trvania aplikácie tak môžu byť matematická fikcia, ktorá nereprezentuje skutočné údaje, na ktorých sa program používa, a najhoršie časy trvania aplikácie môžu byť bizarná konštrukcia, ktorá by sa v praxi nikdy nevyskytla. Po tretie, veľa

programov záujmu nie je dobre pochopených a konkrétne matematické výsledky nemusia byť k dispozícii. Nakoniec, často je to tak, že programy nie sú vôbec porovnateľné: jeden beží oveľa efektívnejšie na jednom konkrétnom druhu vstupu, druhý beží účinne za iných okolností. (Sedgewick, 1990, s. 67 - 68)

Napriek vyššie uvedeným pripomienkam je často možné presne predpovedať, ako dlho bude určitý program trvať, alebo vedieť, že jeden program bude v konkrétnych situáciách pracovať lepšie ako druhý. Úlohou analytika algoritmov je zistiť čo najviac informácií o výkone algoritmov; úlohou programátora je použiť tieto informácie pri výbere algoritmov pre konkrétne aplikácie. (Sedgewick, 1990, s. 68)

Prvým krokom v analýze algoritmu je charakterizácia údajov, ktoré sa majú použiť ako vstup do algoritmu, a rozhodnutie o tom, aký typ analýzy je vhodný. V ideálnom prípade by sme chceli odvodiť pre každé dané rozdelenie pravdepodobnosti výskytu možných vstupov zodpovedajúce rozdelenie možných prevádzkových časov algoritmu. Tento ideál ale nedokážeme dosiahnuť pre žiadny netriviálny algoritmus, preto sa zvyčajne zameriavame na ohraničenie výkonnosti tým, že sa pokúsime dokázať, že prevádzkový čas algoritmu je vždy menší ako nejaký „horný limit“ bez ohľadu na vstup a na odvodenie priemerného času chodu pri „náhodných“ vstupoch. (Sedgewick, 1990, s. 68)

Druhým krokom v analýze algoritmu je identifikácia abstraktných operácií, na ktorých je algoritmus založený, s cieľom oddeliť analýzu od implementácie. Takto napríklad oddelíme štúdiu o tom, koľko porovnávaní robí triediaci algoritmus pri zoraďovaní údajov od toho, koľko mikrosekúnd potrebuje konkrétny počítač na vykonanie akéhokoľvek strojového kódu, ktorý konkrétny kompilátor vytvorí pre určitý fragment kódu. Obidva tieto prvky sú potrebné na určenie skutočnej doby vykonávania programu na konkrétnom počítači. Prvý je určený vlastnosťami algoritmu; ďalší je určený vlastnosťami počítača. Táto separácia nám často umožňuje porovnávať algoritmy, ktoré sú aspoň do istej miery nezávislé od konkrétnych implementácií alebo konkrétnych počítačov. (Sedgewick, 1990, s. 68)

V treťom kroku analýzy algoritmu pristúpime k samotnej matematickej analýze s cieľom nájsť priemerné a najhoršie hodnoty pre každú zo základných veličín. Nie je ťažké nájsť hornú hranicu doby chodu programu - výzvou je nájsť najlepšiu hornú hranicu, ktorú by bolo možné dosiahnuť, keby sa vyskytol najhorší vstup. Tým zistíme najhorší čas trvania. Priemerný čas trvania obvykle vyžaduje dosť sofistikovanú matematickú analýzu. Keď sa takéto analýzy úspešne vykonajú na základných veličinách, je možné určiť čas súvisiaci s každou veličinou a vyjadriť tak celkový čas behu algoritmu. (Sedgewick, 1990, s. 69)

V zásade možno výkonnosť algoritmu často analyzovať na veľmi presnú úroveň detailov, obmedzenú iba neistotou týkajúcou sa výkonu počítača alebo náročnosťou stanovenia matematických vlastností niektorých abstraktných veličín. Zriedkakedy sa oplatí vykonať úplnú podrobnú analýzu, takže nás vždy zaujíma odhad bez zamerania sa na detaily. (V skutočnosti sa odhady, ktoré sa na prvý pohľad zdajú byť veľmi hrubé, často ukazujú ako dosť presné.) Analýza algoritmu je cyklický proces analýzy, odhadu a zdokonalenia analýzy, kým sa nedostaneme na požadovanú úroveň detailov. Tento proces by mal zahŕňať aj zlepšenia v implementácii a takéto zlepšenia sa v skutočnosti v analýze často navrhujú.
(Sedgewick, 1990, s. 69)

2.2 Klasifikácia algoritmov

Ako je uvedené v predchádzajúcej kapitole, väčšina algoritmov má primárny parameter N, zvyčajne počet údajov, ktoré sa majú spracovať. Tento parameter najvýraznejšie ovplyvňuje dobu chodu. Všetky triediace algoritmy spomínané v tejto práci majú čas vykonávania úmerný jednej z nasledujúcich funkcií:

1 Pri tejto funkcii doba chodu algoritmu nezávisí od veľkosti parametra *N* a hovoríme, že je konštantná. Toto je samozrejme situácia, o ktorú sa treba usilovať pri navrhovaní algoritmov. (Sedgewick, 1990, s. 69)

Log N Ak je doba behu programu logaritmická, program sa s rastúcim *N* mierne spomaľuje. Tento čas behu sa bežne vyskytuje v programoch, ktoré riešia veľký problém tak, že ho transformujú na menší problém, zmenšením jeho veľkosti o určitú konštantnú časť. Základ logaritmu mení konštantu, ale nie o veľa. Ak je *N* tisíc, *log N* je 3, ak sa základ logaritmu rovná 10. Ak má základ logaritmu hodnotu 2, hodnota *log N* sa blíži k číslu 10. Ak je *N* milión, *log N* sa zdvojnásobí. Vždy, keď sa *N* zdvojnásobí, *log N* sa zvýši o konštantu, ale *log N* sa nezdvojnásobí, kým sa *N* nezvýši na N^2.
(Sedgewick, 1990, s. 70)

N Keď je doba chodu programu lineárna, vo všeobecnosti platí, že s každým vstupným prvkom sa pridá určité množstvo času spracovania. Ak je *N* milión, zodpovedá tomu aj doba behu. Kedykoľvek sa N zdvojnásobí, zdvojnásobí sa aj doba behu. Toto je optimálna situácia pre algoritmus, ktorý musí spracovať *N* vstupov (alebo vytvoriť *N* výstupov). (Sedgewick, 1990, s. 70)

N log N Táto funkcia platí pre také algoritmy, ktoré riešia problém tak, že ho rozdelia do menších problémov, samostatne ich riešia a potom kombinujú riešenia. Ak je hodnota *N* milión, *N log N* pri základe logaritmu dva je približne dvadsať miliónov.

Keď sa *N* zdvojnásobí, doba chodu algoritmu sa viac ako zdvojnásobí (ale nie oveľa viac). (Sedgewick, 1990, s. 70)

N^2 Ak je doba vykonávania algoritmu kvadratická, je praktické ho použiť iba pri relatívne malých problémoch. Kvadratické prevádzkové doby zvyčajne vznikajú v algoritmoch, ktoré spracúvajú všetky páry údajových položiek (napríklad v dvojitej vnorenej slučke). Ak N je tisíc, doba prevádzky je milión. Kedykoľvek sa N zdvojnásobí, prevádzková doba sa zvýši štvornásobne. (Sedgewick, 1990, s. 70)

2.3 Výpočtová zložitosť

Jedným z prístupov k štúdiu výkonnosti algoritmov je študovať výkonnosť najhoršej doby trvania, ignorujúc konštantné faktory, aby sa určila funkčná závislosť doby chodu (alebo nejakého iného opatrenia) na počte vstupov (alebo nejakej inej premennej). Tento prístup je atraktívny, pretože umožňuje preukázať presné matematické výroky o dobe vykonávania programov. (Sedgewick, 1990, s. 71 - 72)

Prvým krokom v tomto procese je urobiť matematický výklad pojmu „úmerný“, pričom sa súčasne oddelí analýza algoritmu od akejkoľvek konkrétnej implementácie. Cieľom je ignorovať konštantné faktory v analýze. Vo väčšine prípadov, ak chceme vedieť, či je doba vykonávania algoritmu úmerná *N* alebo úmerná *log N*, nezáleží na tom, či sa má algoritmus spustiť na mikropočítači alebo na superpočítači a nezáleží na tom, či bola vnútorná slučka algoritmu starostlivo implementovaná iba s niekoľkými pokynmi alebo zle implementovaná s mnohými pokynmi. Z matematického hľadiska sú tieto dva faktory rovnocenné. Matematický výraz na spresnenie tohto pojmu sa nazýva O-notácia (notácia Omikron) alebo „Landauova notácia“. (Sedgewick, 1990, s. 72)

Množina Omikron funkcie *f* je množina všetkých funkcií, ktoré majú "rovnakú" alebo "nižšiu" rýchlosť rastu ako funkcia f. Ak prehlásime, že zložitosť algoritmu leží v množine Omikron funkcie *f*, znamená to, že náročnosť algoritmu bude rásť vždy rovnako alebo pomalšie ako funkcia *f*. Notáciu Omikron teda možno chápať ako vyjadrenie "najhoršej doby chodu". (Hordějčuk, 2008)

Tvrdenie, že doba chodu algoritmu je O (f (N)), je nezávislé na vstupe algoritmu. Pretože sa zaujímame o štúdium algoritmu, nie o vstup alebo implementáciu, notácia O je užitočný spôsob, ako určiť hornú hranicu doby chodu, ktorá je nezávislá od vstupov aj od podrobností o implementácii. Pri interpretácii výsledkov vyjadrených pomocou O-notácie musíme byť veľmi opatrný a to okrem iného aj z týchto dôvodov: po prvé, je to „horná hranica“

a skutočné výsledky môžu byť oveľa nižšie; po druhé, vstup, ktorý spôsobuje najhorší čas chodu aplikácie, sa pravdepodobne nikdy nebude vyskytovať v praxi.
(Sedgewick, 1990, s. 72 - 73)

3 Triediace algoritmy

V praxi sú čísla, ktoré sa majú triediť len zriedka izolované hodnoty. Väčšinou sú tieto čísla súčasťou súboru dát nazývaného záznam. Každý záznam obsahuje kľúčový atribút, čo je hodnota, ktorá sa má zoraďovať. Zvyšok záznamu tvoria tzv. satelitné údaje, ktoré spolu s kľúčovou hodnotou tvoria záznam. Kľúčové atribúty, ktoré sú iba časťou záznamov (často malou časťou), sú väčšinou hodnotou, podľa ktorej sa záznamy usporadúvajú. Cieľom metódy triedenia je zmeniť usporiadanie záznamov tak, aby sa ich kľúče zoradili podľa nejakého dobre definovaného pravidla usporiadania (zvyčajne číselného alebo abecedného poradia). V praxi, keď triediaci algoritmus permutuje kľúčové atribúty, musí tiež permutovať aj celé záznamy. Ak každý záznam pozostáva z veľkého množstva údajov, často permutujeme len rad ukazovateľov na záznamy radšej ako samotné záznamy, aby sa minimalizoval pohyb s údajmi. (Cormen a kol., 2009, s. 147)

Transformácia algoritmu na triedenie čísel do programu na triedenie záznamov je koncepčne jednoduchá, aj keď v danej technickej situácii môžu určité detaily spôsobiť, že samotné naprogramovanie daného programu bude výzvou.
(Cormen a kol., 2009, s. 147 - 148)

Pred spomenutím niektorých konkrétnych triediacich algoritmov bude užitočné prediskutovať všeobecnú terminológiu a základné predpoklady pre triediace algoritmy. Hlavným parametrom výkonnosti, ktorý nás bude zaujímať, je doba chodu triediacich algoritmov. Druhou charakteristikou triediacich metód, ktorá je v praxi niekedy dôležitá, je stabilita. Metóda triedenia sa nazýva stabilná, ak zachováva relatívne poradie rovnakých kľúčov v súbore. Napríklad, ak chceme abecedne usporiadaný zoznam študentov v triede usporiadať podľa známky, potom stabilná metóda vytvorí zoznam, v ktorom sú študenti s rovnakou známkou stále v abecednom poradí, ale nestabilná metóda pravdepodobne vytvorí zoznam bez pozostatkov pôvodného abecedného zoznamu. (Sedgewick, 1990, s. 94)

Aby sme sa mohli sústrediť na algoritmické problémy, budeme pracovať s algoritmami, ktoré jednoducho triedia polia celých čísel do číselného poradia. Spravidla je jednoduché takéto algoritmy prispôsobiť na použitie v praktickej aplikácii zahŕňajúcej veľké množstvo kľúčových atribútov alebo záznamov. V zásade triediace programy pristupujú

k záznamom jedným z dvoch spôsobov: k porovnávaniu sú prístupné buď len kľúčové atribúty, alebo sa presúvajú celé záznamy. (Sedgewick, 1990, s. 95)

3.1 Prečo je triedenie potrebné?

Mnoho počítačových vedcov považuje triedenie za najzákladnejší problém v štúdii algoritmov. Existuje niekoľko dôvodov:

- Niekedy aplikácia nevyhnutne potrebuje utriediť informácie. Napríklad v záujme prípravy výkazov pre klientov musia banky zoradiť šeky podľa kontrolných čísiel.
- Algoritmy často používajú triedenie ako kľúčový podprogram. Napríklad program, ktorý vykresľuje grafické objekty, ktoré sú navrstvené nad sebou, by mohol objekty zoradiť podľa toho, ako sú nad sebou, aby mohol tieto objekty nakresliť zdola nahor.
- Môžeme čerpať zo širokej škály triediacich algoritmov a z ich bohatej škály techník, ktoré na triedenie používajú. V skutočnosti sa mnoho dôležitých techník používaných pri navrhovaní algoritmov objavuje aj v tele triediacich algoritmov, ktoré boli vyvinuté v priebehu niekoľkých rokov. Týmto spôsobom je triedenie tiež problémom historického záujmu.
- Pri zavádzaní triediacich algoritmov sa do popredia dostáva veľa technických problémov. Najrýchlejší zoraďovací program pre konkrétnu situáciu môže závisieť od mnohých faktorov, ako sú napríklad predchádzajúce znalosti o kľúčových a satelitných údajoch, hierarchia pamäte (cache a virtuálna pamäť) hostiteľského počítača a softvérové prostredie. Mnohé z týchto problémov je najlepšie riešiť už na algoritmickej úrovni, a nie „vylaďovaním“ kódu.

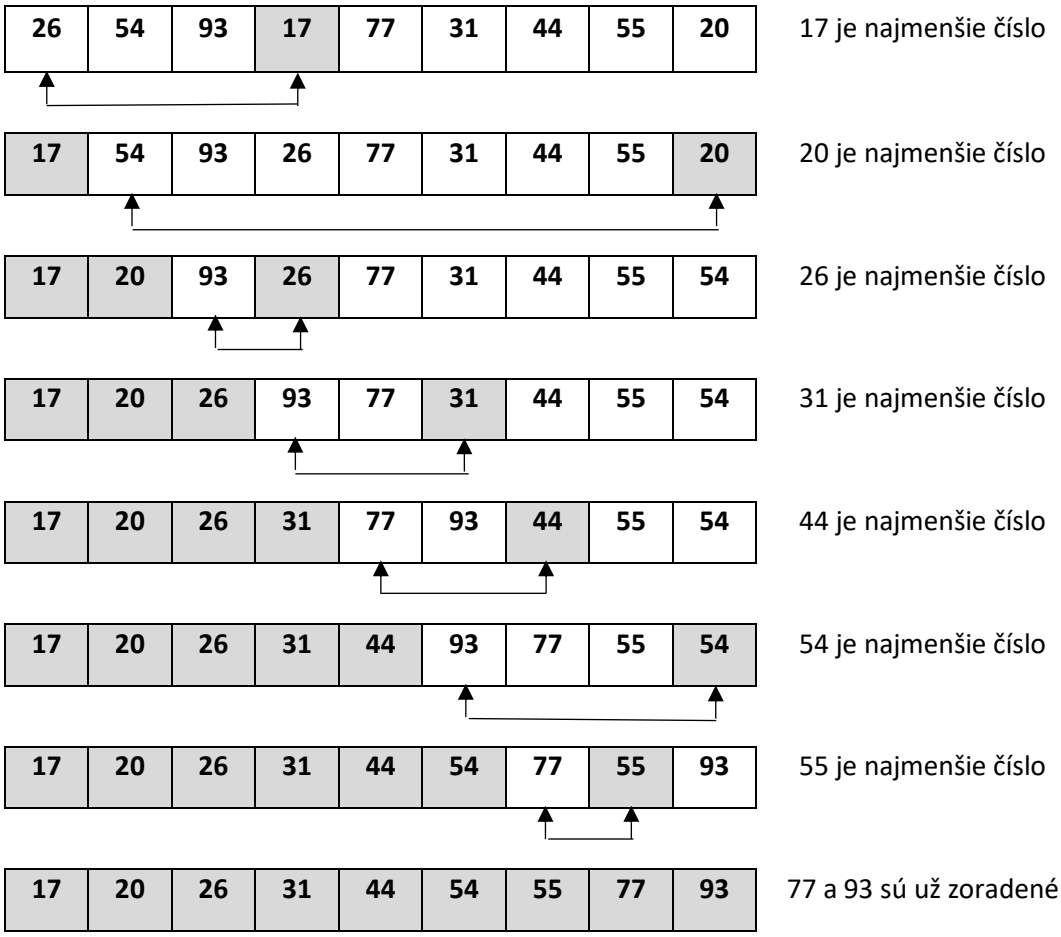
(Cormen a kol., 2009, s. 148)

3.2 Selection Sort

Selection sort je jeden z najjednoduchších algoritmov triedenia funguje a nasledovne: Najprv nájde najmenší prvok v poli a vymení ho s prvkom na prvej pozícii, potom nájde druhý najmenší prvok a vymení ho s prvkom na druhej pozícii a pokračuje v tom, až kým nezoradí celé pole. Táto metóda sa nazýva Selection sort (výberové triedenie), pretože funguje opakovaným „výberom“ najmenšieho zostávajúceho prvku, ako je znázornené na obrázku č. 1, ktorý zobrazuje celý proces. (Sedgewick, 1990, s. 96)

Každým prechodom cez pole je vybraný najmenší zostávajúci prvok a je premiestnený na správne miesto. Prvý prechod presunie číslo 17 na prvú pozíciu, pretože v poli nie je žiadny prvok menší ako 17. V druhom prechode je druhý najmenší zostávajúci prvok 20,

takže je vymenená za 54 na druhej pozícii. Potom je 26 v poli vymenená za 93 na tretej pozícii pri treťom prechode, druhé 31 je vymenené za 93 na štvrtej pozícii pri štvrtom prechode atď.



Obrázok č. 1 – Grafické znázornenie algoritmu Selection sort

(vlastné spracovanie podľa predlohy Miller a Ranum, 2005, s. 168)

Nasledujúci kód je implementáciou tohto procesu v jazyku C. Pre každé *i* od 0 až po *N-1* vymení *a[i]* za minimálny prvok v *a[i],...,a[N]*:

```
void selectionSort(int a[], int N)
{
   int i, j , min, t;
   for (i = 0; i < N; i++)
     {
     min = i;
     for (j = i+1; j < N; j++)
         if (a[j] < a[min]) min = j;
     t = a[min]; a[min] = a[i]; a[i] = t;
     }
}
```

svojej konečnej pozícii v poli (a už sa ich nebude znova dotýkať), takže pole sa úplne zoradí, keď sa index dostane na pravú stranu poľa. „Vnútorná slučka“ slúži na porovnanie *a[j]* < *a[min]*. Keďže sa index *j* s každou ďalšou iteráciou zvyšuje, algoritmus postupne porovná všetky prvky s prvokom *a[min]* a ak nájde prvok menší ako *a[min]*, uloží jeho pozíciu do premennej *min*. Po prejdení celého poľa vymení najmenší nájdený prvok za prvok *a[i]*, čo je v súčasnej iterácii prvým prvkom naľavo v nezoradenej časti poľa. Táto metóda triedenia funguje veľmi dobre pre malé súbory dát.
(Sedgewick, 1990, s. 98)
Časová zložitosť algoritmu Selection sort je $O(N^2)$. (Miller a Ranum, 2005, s. 167)

3.3 Insertion sort

Algoritmus takmer rovnako jednoduchý ako výberové usporiadanie, ale možno flexibilnejší, je Insertion sort. Túto metódu ľudia často používajú na zoradenie kariet: zvažujú jednotlivé karty jednu po druhej a vkladajú každú kartu na svoje správne miesto medzi tie, ktoré už predtým brali do úvahy. (Sedgewick, 1990, s. 98)

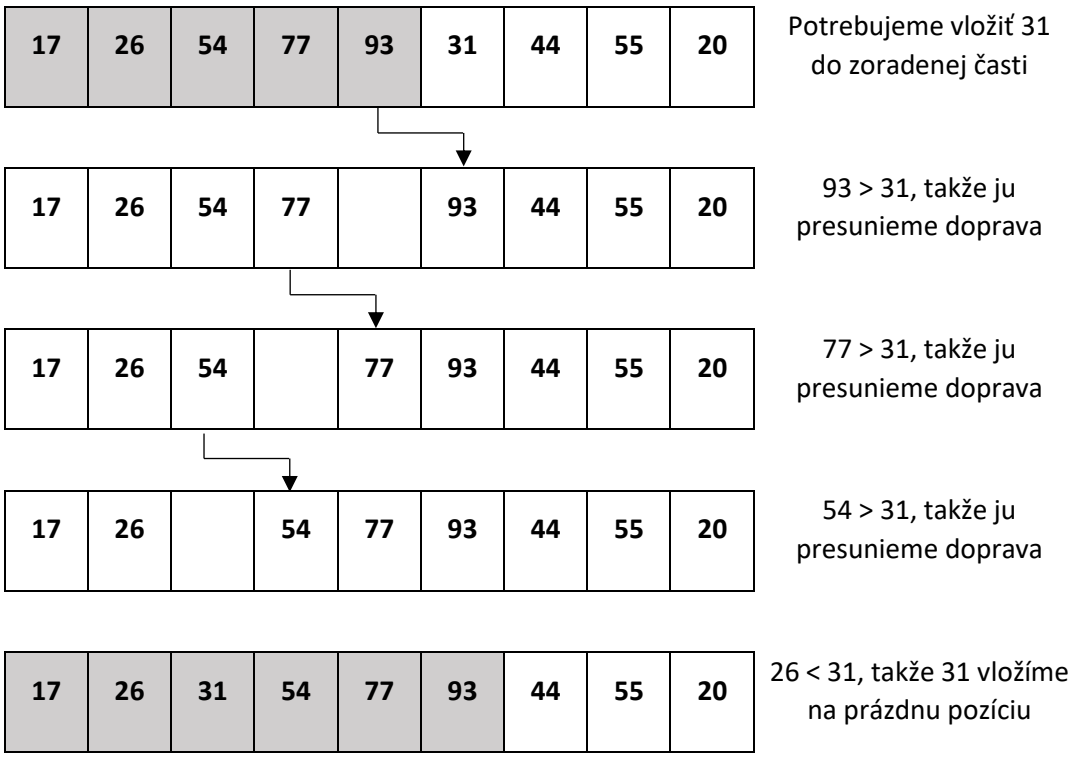
Porovnávaný prvok sa vkladá už medzi zoradené prvky tak, že väčšie prvky sa presunú o jednu pozíciu doprava a následne sa prvok vloží na uvoľnenú pozíciu, tak ako je znázornené na obrázku č. 2. Číslo 26 na druhej pozícii je menšie ako 54, takže 54 sa posunie o jednu pozíciu doprava a na jeho pôvodnú pozíciu sa presunie 26. 93 je väčšia ako 54, takže sa presúvať nemusí a tak sa postupne zoradia prvky do správneho poradia.

54	**26**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	Prvý usporiadaný prvok
26	**54**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	vložená 26
26	**54**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	vložená 93
17	**26**	**54**	**93**	**77**	**31**	**44**	**55**	**20**	vložená 17
17	**26**	**54**	**77**	**93**	**31**	**44**	**55**	**20**	vložená 77
17	**26**	**31**	**54**	**77**	**93**	**44**	**55**	**20**	vložená 31
17	**26**	**31**	**44**	**54**	**77**	**93**	**55**	**20**	vložená 44
17	**26**	**31**	**44**	**54**	**55**	**77**	**93**	**20**	vložená 55
17	**20**	**26**	**31**	**44**	**54**	**55**	**77**	**93**	vložená 20

Obrázok č. 2 – Grafické znázornenie algoritmu Insertion sort

(Miller a Ranum, 2005, s. 170)

Obrázok č. 3 zobrazuje podrobnejšie piatu iteráciu. V tomto bode algoritmu máme už zotriedenú časť poľa pozostávajúcu z prvkov 17, 26, 54, 77 a 93. Chceme vložiť 31 späť do už triedených položiek. Prvé porovnanie s 93 má za následok, že sa 93 posunie doprava. 77 a 54 sa tiež posunú, pretože sú väčšie ako číslo 31. Keď sa objaví prvok 26, proces radenia sa zastaví a 31 sa umiestni do otvorenej polohy. Teraz máme v zotriedenej časti poľa šesť položiek. (Miller a Ranum, 2005, s. 167)



Obrázok č. 3 – Piata iterácia triedenia pomocou Insertion sortu

(Miller a Ranum, 2005, s. 171)

Nasledujúcim kódom sa tento proces implementuje v jazyku C. Pre každé *i* od *1* do *N* sa prvky *a [1], ..., a [i]* zoraďujú umiestnením *a[i]* na miesto v zoradenej časti zoznamu prvkov *a[1] ..., a[i-1]*:

```
void insertionSort(int a[], int N)
{
    int i, v, j;
    for (i = 1; i < N; i++)
     {
        v = a[i];
        j = i - 1;
        while (j >= 0 && a[j] > v)
        {

            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = v;
    }
}
```

Rovnako ako pri Selection sorte sú prvky naľavo od indexu *i* počas procesu zoradené, ale nie sú vo svojej konečnej polohe, keďže sa môžu presunúť, aby sa uvoľnili priestor pre menšie prvky. Keď sa však index dostane na pravý koniec poľa, pole je úplne zoradené. Slučka *while* porovnáva aktuálny prvok so všetkými prvkami naľavo. Všetky prvky väčšie ako *a[i]* presunie o jednu pozíciu doprava. Posledný riadok slučky *for* následne zaradí aktuálny prvok *a[i]* pred nich. (Sedgewick, 1990, s. 100)

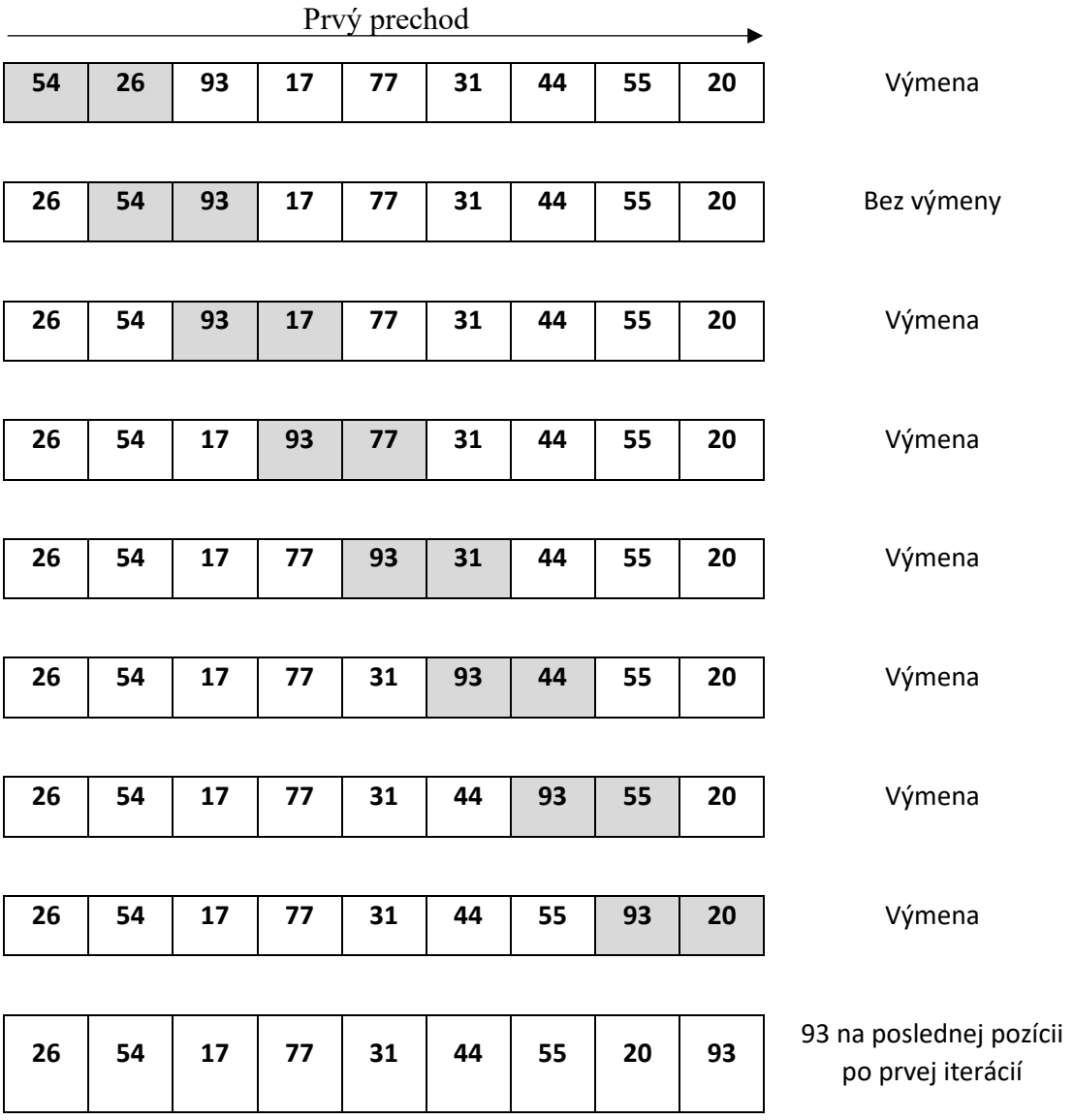
Časová zložitosť algoritmu Insertion sort v najhoršom prípade je $O(N^2)$, to je vtedy, ak sú všetky prvky, ktoré sa majú usporiadať v opačnom poradí. V najlepšom prípade je však potrebné vykonať pri každom prechode iba jedno porovnanie. To by bol prípad už zoradených prvkov. (Miller a Ranum, 2005, s. 169)

Insertion sort je stabilný triediaci algoritmus, ktorý sa používa, ak je počet prvkov v poli malý. Môže byť užitočný aj vtedy, ak je vstupné pole už takmer zoradené a treba usporiadať len niekoľko prvkov vo veľkom poli. (Nayal a kol., 2013)

3.4 Bubble sort

Bubble sort prechádza cez pole niekoľko krát. Porovnáva susediace prvky a vymieňa tie, ktoré sú v nesprávnom poradí. Každý prechod poľom umiestni ďalšiu najväčšiu hodnotu na svoje správne miesto. V podstate každý prvok „prebublá“ až na miesto, kam patrí. (Miller a Ranum, 2005, s. 164)

Obrázok č. 4 zobrazuje prvý prechod Bubble sortu. Zvýraznené prvky sa porovnávajú, aby sa zistilo, či sú v správnom poradí. Ak je v poli *N* prvkov, potom existuje *N-1* párov prvkov, ktoré je potrebné porovnať pri prvom prechode. Je dôležité si uvedomiť, že akonáhle je najväčšia hodnota v poli súčasťou páru, bude sa nepretržite pohybovať až na koniec poľa, dokiaľ nie je prechod dokončený. (Miller a Ranum, 2005, s. 164)



Obrázok č. 4 – Prvá iterácia algoritmu Bubble sort (Miller a Ranum, 2005, s. 165)

Na začiatku druhého prechodu je teraz najvyššia hodnota na správnej pozícii. V poli zostáva *N-1* položiek na zoradenie, čo znamená, že párov, ktoré sa budú porovnávať bude *N-2*. Pretože každý prechod umiestni najväčší zostávajúci prvok na svoje miesto, celkový počet potrebných prechodov bude *N-1*. Po dokončení *N-1* prechodov musí byť najmenší prvok umiestnený na správnej pozícii bez vyžadovania ďalšieho procesu.

Implementácia Bubble sort v jazyku C:

```
void bubbleSort(int a[], int N)
{
int i, j, t;
for (i = N; i > 1; i--)
for (j = 1; j < i; j++)
if (a[j-1] > a[j])
{ t = a[j-1]; a[j-1] = a[j]; a[j] = t; }
```

Aby sme mohli Bubble sort zanalyzovať, mali by sme poznamenať, že bez ohľadu na to, ako sú prvky usporiadané v pôvodnom poli, na zoradenie poľa s veľkosťou *N* prvkov bude potrebných *N-1* prechodov poľom. V najlepšom prípade, ak je zoznam už na začiatku správne zoradený, nebudú uskutočňované žiadne výmeny. V najhoršom prípade však každé jedno porovnanie spôsobí výmenu. (Miller a Ranum, 2005, s. 165)

Bubble sort sa často považuje za najmenej účinný spôsob triedenia, pretože musí vymieňať položky skôr, ako bude známe ich konečné umiestnenie. Tieto „zbytočné“ devízové operácie sú veľmi nákladné. Pretože však Bubble sort prechádza celou netriedenou časťou poľa, má schopnosť urobiť niečo, čo nedokáže väčšina triediacich algoritmov. Najmä, ak počas prechodu nedochádza k žiadnym výmenám, potom vieme, že zoznam už musí byť usporiadaný. Bubble sort je možné upraviť tak, aby sa čo najskôr zastavil, ak zistí, že zoznam už je utriedený. To znamená, že pre zoznamy, ktoré vyžadujú iba niekoľko prechodov, môže mať Bubble sort tú výhodu, že rozpozná zoradený zoznam a zastaví sa. Časová zložitosť pri najhoršej dobe chodu je $O(N^2)$.
(Miller a Ranum, 2005, s. 165- 166)

Vďaka svojej jednoduchosti sa Bubble sort často používa na predstavenie pojmu triediaci algoritmus. V počítačovej grafike je populárny pre svoju schopnosť detegovať veľmi

malú chybu (ako je výmena iba dvoch prvkov) v takmer zoradených poliach a opraviť ju iba lineárnou komplexnosťou (2N). (Nayal a kol, 2014)

3.5 Shaker sort (Cocktail sort)

Na dôkladné zanalyzovanie Bubble sortu bolo potrebné vynaložiť veľa práce. Hoci techniky použité pri výpočtoch sú prínosné a zaujímavé, výsledky sú sklamaním, pretože nám hovoria, že Bubble sort nie je vôbec veľmi dobrý. V porovnaní s Insertion sortom vyžaduje Bubble sort komplikovanejší program a trvá viac ako dvakrát tak dlho.

Niektoré z nedostatkov Bubble sortu je možné ľahko zistiť. Jedným z nich je napríklad aj to, že prvky sa nikdy nemôžu posunúť doľava o viac ako o jednu pozíciu pri jednom prechode poľom, takže ak je najmenšia položka v nezoradenom poli úplne vpravo, algoritmus je nútený urobiť maximálny počet porovnaní. Aj práve preto vznikla vylepšená verzia Bubble sortu z názvok Shaker sort (tiež Cocktail sort), ktorá prechádza cez pole striedavo oboma smermi. (Knuth, 1998, s. 109 - 110)

Shaker sort je variáciou triediaceho algoritmu Bubble sort. Algoritmus Bubble sort vždy prechádza prvky zľava a pri prvej iterácii posúva najväčší prvok na svoju správnu pozíciu, druhý najväčší prvok zoraďuje v druhej iterácii atď. Shaker sort prechádza striedavo daným poľom v oboch smeroch.

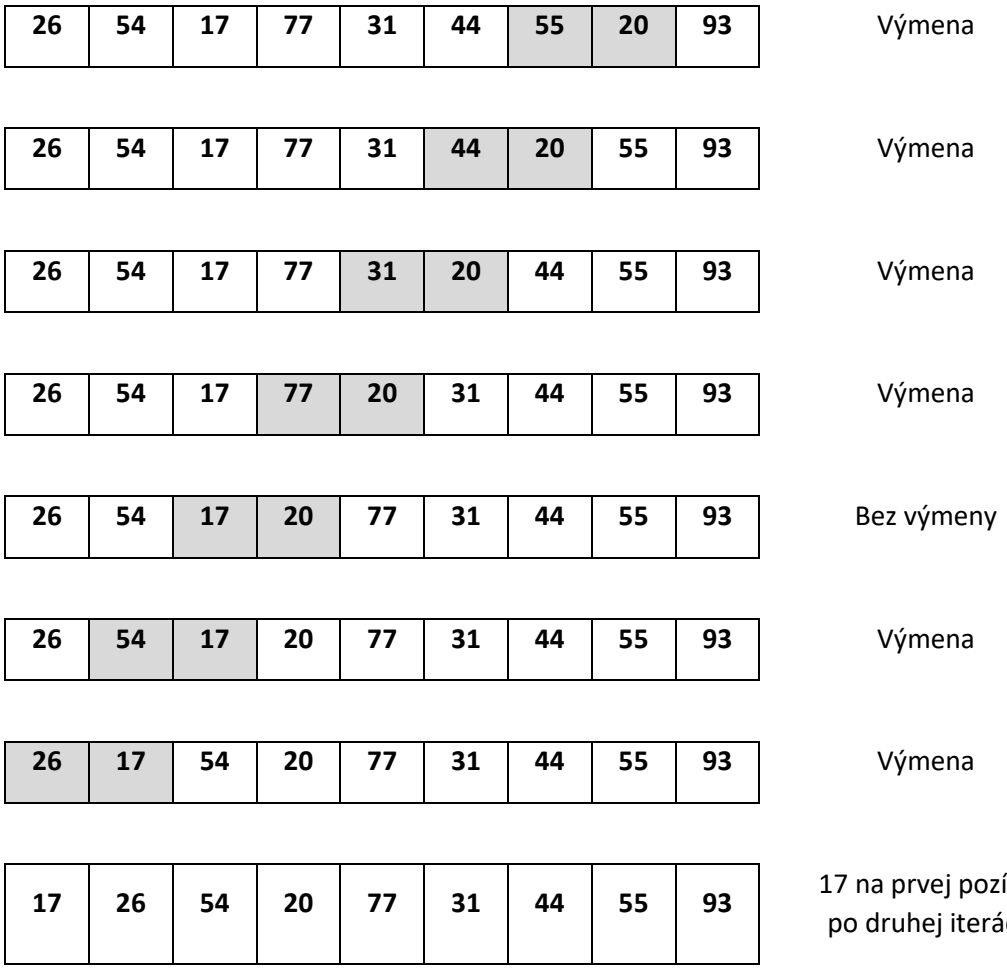
Každá iterácia algoritmu je rozdelená do dvoch stupňov:

1) Prvá etapa prechádza cez pole zľava doprava, rovnako ako Bubble sort. Počas slučky sa susediace prvky porovnávajú a ak je hodnota vľavo väčšia ako hodnota vpravo, hodnoty sa prehodia. Na konci prvej iterácie bude najväčší prvok na konci poľa.

2) Druhá fáza prechádza cez pole v opačnom smere - počnúc prvkom tesne pred naposledy zoradeným (najväčším) prvkom sa pohybuje späť na začiatok poľa. Aj v tejto fáze sa porovnávajú susedné položky a podľa potreby sa vymieňajú. (Sahai, 2019)

Prvý prechod poľom je totožný s prechodom algoritmu Bubble sort znázorneným na obrázku č. 4. Na obrázku č. 5 je znázornený druhý prechod poľom algoritmu Shaker sort sprava doľava. Keďže prvok 93 bol umiestnený na svoje miesto už pri prvom prechode, porovnávanie sa začína od prvku číslo 20, čo je prvok najviac vpravo v ešte nezotriedenej časti poľa. Algoritmus porovnáva zvýraznené dvojice a podľa potreby zamieňa ich pozície. Môžeme si

všimnúť, že prvok 20 sa už v druhom prechode poľom presunul o päť pozícií doprava, pretože v pôvodnom nezoradenom poli sa nachádzal na poslednej pozícii, zatiaľ čo algoritmu Bubble sort by takýto presun malého prvku trval päť prechodov poľom. (vlastné spracovanie)



Obrázok č. 5 – Druhý prechod poľom algoritmu Shaker sort (vlastné spracovanie)

Implementácia Shaker sortu v jazyku C:

```
#define vymenCisla(a, b) {long int tmp = a; a = b; b = tmp; }

void shakeSort( int pole[], int vstup)
{
      int lavy = 1, pravy = vstup - 1, i, k;
      do {
            for (i = pravy; i >= lavy; i--)
                  if (pole[i - 1] > pole[i])
                  {
                        vymenCisla(pole[i - 1], pole[i]);
                        k = i;
                  }
            lavy = k + 1;
            for (i = lavy; i <= pravy; i++)
                  if (pole[i - 1] > pole[i])
                  {
                        vymenCisla(pole[i - 1], pole[i]);
                        k = i;
                  }
            pravy = k - 1;
      } while (lavy < pravy);
}
```

(Zdroj: prednáška Triediace (usporiadavacie) algoritmy v C++ programe Ing. Igora Košťála, PhD.)

Algoritmus v tejto implementácii začína s porovnávaním dvoch susediacich prvkov najviac napravo. Ak je prvok *pole[i-1]* väčší ako *pole[i],* vymení ich pozície pomocou definovaného makra *vymenCisla*. Index *i* sa každou iteráciu zmenšuje a algoritmus pokračuje v porovnávaní dvojíc prvkov, až pokiaľ sa nedostane na ľavý koniec poľa, kam umiestni najmenší prvok v poli. Následne sa začne vracať naspäť na pravú stranu poľa a takto pokračuje, kým celé pole nie usporiadané. Celočíselné premenné *lavy* a *pravy*, ktoré udávajú ľavý a pravý koniec nezoradenej časti poľa postupne zužujú medzeru medzi nimi, aby algoritmus

neporovnával čísla už v zoradenej časti poľa. Algoritmus skončí vtedy, keď premenná *lavy* bude menšia ako premenna *pravy*. (vlastné spracovanie)

Porovnanie s Bubble sort

Napriek tomu, že časová zložitosť oboch algoritmov je rovnaká ($O(N^2)$), má Shaker sort lepšiu výkonnosť ako Bubble Sort. Zvyčajne je Shaker sort približne dvakrát rýchlejší ako Bubble sort. Pretože Shaker sort pole prechádza obojsmerne, počet porovnávaní sa zredukuje, čím sa mierne zníži celkový čas chodu. (Sahai, 2019)

3.6 Shell sort

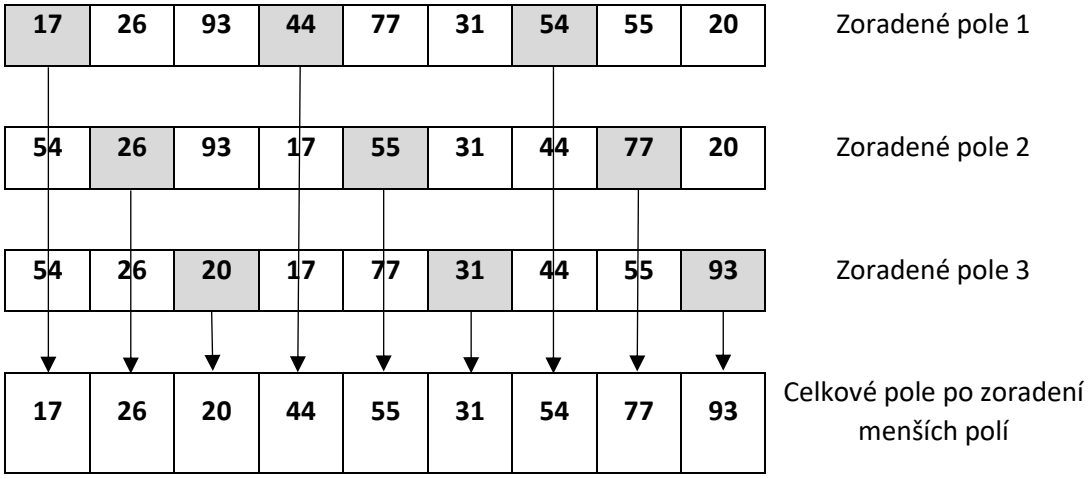
Zoradenie poľa pomocou Insertion sortu je pomalé, pretože vymieňa iba susediace prvky. Ak sa napríklad stane, že najmenší prvok sa vyskytne na konci poľa, je potrebných N krokov, aby sa dostal tam, kam patrí. Shell sort je jednoduché rozšírenie Insertion sortu, ktoré zvyšuje rýchlosť tým, že umožňuje výmenu prvkov, ktoré sú veľmi vzdialené. (Sedgewick, 1990, s. 107)

Shell sort upraví pôvodné pole na niekoľko menších polí, z ktorých každý je zoraďované pomocou Insertion sortu. Jedinečný spôsob, akým sú tieto menšie polia vytvárané, je kľúčom k Shell sortu. Namiesto toho, aby sa pole rozdelilo na menšie polia susediacich prvkov, algoritmus Shell sort používa tzv. prírastok *i*, niekedy nazývaný aj medzera, na vytvorenie polí výberom všetkých prvkov, ktoré sú od seba vo vzdialenosti *i* prvkov.

Takéto rozdelenie poľa na menšie polia môžeme vidieť na obrázku č. 6. Znázornené pole obsahuje deväť prvkov. Ak použijeme prírastok $i = 3$, vzniknú nám tri menšie polia, z ktorých každé môže byť usporiadaný Insertion sortom. Po dokončení usporiadania týchto polí vznikne pole zobrazené na obrázku č. 7. Hoci tento zoznam nie je úplne zoradený, stalo sa niečo veľmi zaujímavé. Zoradením menších polí sme prvky presunuli bližšie k miestu, kde skutočne patria. (Miller a Ranum, 2005, s. 171)

54	**26**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	Pole 1
54	**26**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	Pole 2
54	**26**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	Pole 3

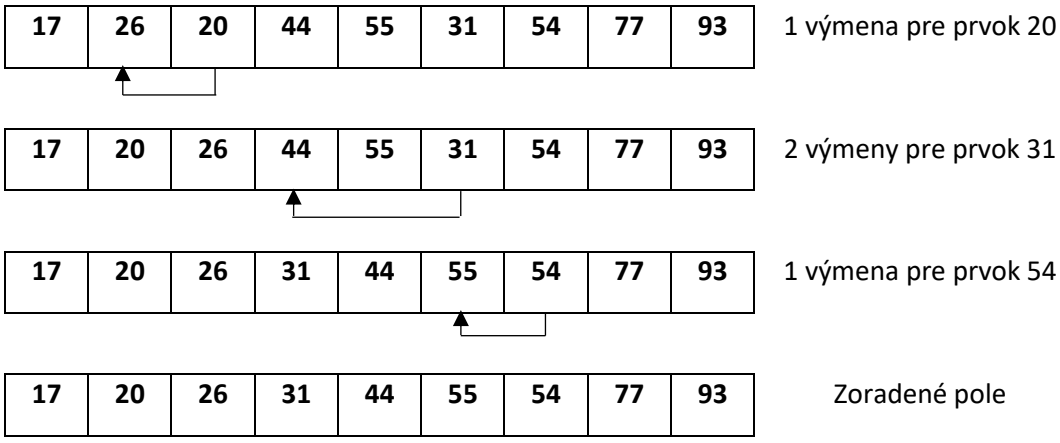
Obrázok č. 6 – Shell sort s prírastkom i = 3 (Miller a Ranum, 2005, s. 172)



Obrázok č. 7 – Shell sort po zoradení každého z menších polí

(Miller a Ranum, 2005, s. 172)

Obrázok č. 8 zobrazuje konečné usporiadanie pomocou prírastku *i = 1*; inými slovami, pomocou štandardného Insertion sortu. Môžeme vidieť, že vykonaním predchádzajúcom usporiadaní menších polí sme teraz znížili celkový počet výmen potrebných na usporiadanie poľa do jeho konečného poradia. V tomto prípade potrebujeme iba štyri ďalšie výmeny na dokončenie procesu. (Miller a Ranum, 2005, s. 171)



Obrázok č. 8 – Shell sort: Konečné usporiadanie poľa pomocou prírastku *i = 1*

(Miller a Ranum, 2005, s. 172)

Zoraďovaním poľa pomocou veľkých prírastkov *i* môžeme presúvať prvky na veľkú vzdialenosť, čím uľahčujeme zoraďovanie poľa pomocou menších prírastkov *i*. Použitím takéhoto postupu pre akúkoľvek postupnosť hodnôt prírastkov *i*, ktorá končí 1 a pomocou ktorého sa pole zoradí sa nazýva Shell sort.

Jedna z veľmi efektívnych prírastkových sekvencií je ..., 1093, 364, 121, 40, 13, 4, 1. Je pravda, že mnoho ďalších prírastkových sekvencií vedie k efektívnejšiemu triedeniu ale je ťažké poraziť vyššie uvedenú sekvenciu o viac ako 20%, dokonca aj pre relatívne mnohoprvkové polia. (Možnosť, že oveľa lepšie prírastkové sekvencie existujú, je stále celkom reálna.) V praxi by sa mohli vyskytnúť aj ďalšie prírastkové sekvencie, pri ich výbere je ale potrebné postupovať opatrne. Na druhej strane existujú aj niektoré zlé prírastkové sekvencie: napríklad 64, 32, 16, 8, 4, 2, 1 pravdepodobne povedie k zlej výkonnosti, pretože prvky na nepárnych pozíciách sa až do konca neporovnávajú s prvkami na párnych pozíciách. Podobne je Shell sort niekedy implementovaný aj určením prírastku, ktorý začína v $i = N$. To doslova zaisťuje, že sa u niektorých N objaví zlá sekvencia. Vyššie uvedený popis účinnosti Shell sort je nevyhnutne nepresný, pretože nikto nebol schopný analyzovať tento algoritmus. To sťažuje nielen vyhodnotenie rôznych prírastkových sekvencií, ale aj analytické porovnanie Shell sort s inými metódami. Nie je známa ani funkčná forma doby behu pre Shell sort (forma navyše závisí od prírastkovej postupnosti).

Na základe vyššie uvedeného správania sa dá povedať, že zložitosť Shell sortu spadá niekde medzi O(N) a $O(N^2)$. Pre prírastok i uvedený vyššie je zložitosť $O(N^2)$. Zmenou prírastku, napríklad použitím $2^k - 1$ (1, 3, 7, 15, 31 atď.) sa môže Shell sort vykonať pri časovej zložitosti $O(N^{\frac{3}{2}})$. (Sedgewick, 1990, s. 110)

Implementácia Shell sortu v jazyku C:

```
#define vymenCisla(a, b) {long int tmp = a; a = b; b = tmp; }

void shellSort(int pole[], int vstup)
{
      int m = vstup;
      while (m /= 2)
      for (int d = vstup - m, i = 0; i < d; i++)
      for (int j = i; (j >= 0) && (pole[j] > pole[j+m]); j -= m)
            vymenCisla(pole[j], pole[j + m]);
}
```

(Zdroj: prednáška Triediace (usporiadavacie) algoritmy v C++ programe Ing. Igora Košťála, PhD.)

Pri tejto implementácii sa nadstaví počiatočný prírastok *i* (v tomto prípade celočíselná premenná *m*) na polovičnú veľkosť poľa a postupne sa vždy o polovicu znižuje, až kým nedosiahne hodnotu 0. Vtedy sa už ďalšia iterácia neopakuje a pole je usporiadané.

Na prvý pohľad sa môže zdať, že Shell sort nemôže byť rýchlejší a lepší ako Insertion sort, pretože ako posledný krok vykonáva kompletné usporiadanie pomocou Insertion sortu. Ukazuje sa však, že toto konečné vykonanie Insertion sortu nemusí robiť veľmi veľa porovnávaní (alebo výmen), pretože pole bolo vopred usporiadané pomocou rozdelenia na menšie časti, podľa určeného prírastku *i* tak, ako je opísané vyššie. Inými slovami, každý prechod vytvorí pole, ktoré je „viac usporiadané“ ako predchádzajúce. Vďaka tomu je posledný prechod veľmi efektívny. (Miller a Ranum, 2005, s. 173)

Shell sort je metóda, ktorá je vhodná pre mnoho triediacich aplikácií, pretože má prijateľnú prevádzkovú dobu aj pre väčšie súbory a vyžaduje iba veľmi malé množstvo kódu, s ktorým sa ľahko pracuje. (Sedgewick, 1990, s. 111)

4 Implementácia algoritmov Shake sort a Shell sort v programe vytvorenom v jazyku C

V tejto kapitole ukážeme, ako efektívne fungujú triediace algoritmy Shake sort a Shell sort v programe vytvorenom v programovacom jazyku C pri triedení veľkých celočíselných polí s niekoľko stotisíc prvkami. Cieľom je vytvoriť program, do ktorého používateľ zadá veľkosť poľa čísel, ktoré sa má usporadúvať. Toto pole môže obsahovať až stotisíce prvkov. Následne program vytvorí dynamické pole s veľkosťou, ktorú používateľ zadal, pomocou generátora pseudonáhodných čísel vygeneruje a vloží celé čísla do daného poľa. Chceme, aby vygenerované čísla boli čo najväčšie, s hodnotami až do dvoch miliárd. Ďalej toto pole zoradí vybranými algoritmami Shake sort a Shell sort vzostupne aj zostupne, zmeria ich exekučné časy a pomocou testovacej funkcie zistí, či bolo pole zoradené správne. Ak test prebehne úspešne, výsledky exekučných časov program zapíše do výsledného súboru s časovou pečiatkou. Výsledky porovnáme a zistíme, ktorý zo spomínaných algoritmov je efektívnejší pre usporiadanie čo najväčšieho celočíselného poľa, aké nám náš počítač umožní vytvoriť. Algoritmy budeme spúšťať a porovnávať na počítači so 64-bitovou verziou operačného systému Windows 10, 8 GB RAM a procesorom AMD FX-6300. (vlastné spracovanie)

4.1 Načítanie veľkosti poľa od používateľa

Po vytvorení nového projektu vo vývojom prostredí Visual Stuido 2019 prejdeme k tvorbe programu. Prvým krokom bude vytvorenie poľa, do ktorého program pomocou generátora pseudonáhodných čísel vloží vygenerované čísla. Chceme, aby používateľ programu zadal veľkosť poľa celých čísel. Aby sme mohli od používateľa načítať vstup z klávesnice, potrebujeme do nášho zdrojového kódu zahrnúť knižnicu *stdio.h* pomocou direktívy preprocesora „*#include <stdio.h>*“. Táto knižnica okrem iného obsahuje aj deklaráciu funkcie *scanf*, ktorá sa používa na čítanie znakov, reťazcov alebo čísel z konzoly. Ďalej obsahuje deklaráciu funkcie *printf*, ktorú budeme používať na zobrazenie znakov, reťazcov alebo čísel na konzolu. Ďalšie funkcie, ktorých deklarácie táto knižnica obsahuje a ktoré budeme používať spomenieme neskôr. Do obidvoch zdrojových súborov *main.c* a *funkcie.c*, ktoré sme v projekte vytvorili, zahrnieme aj vytvorený hlavičkový súbor *deklaracie.h*, kde budeme neskôr pridávať deklarácie vytvorených funkcií. (vlastné spracovanie)

Najskôr si musíme definovať premennú, do ktorej neskôr vložíme takú veľkosť poľa, akú načítame od používateľa. Keďže od používateľa očakávame, že bude zadávať veľkosť poľa v stotisícoch, použijeme dátový typ *long int*. Je to celočíselný dátový typ, do ktorého je možné uložiť hodnotu z rozsahu −2 147 483 647 – 2 147 483 647. (Zdroj: ISO/IEC 9899:2011, s. 27). Premennú nazveme „*vstup*“ a definujeme ju príkazom „*long int vstup;*“.

Teraz môžeme od používateľa načítať vstup a uložiť ho do vytvorenej premennej. Najskôr ho vyzveme pomocou funkcie *printf*, vďaka ktorej vypíšeme na konzolu výzvu, aby zadal veľkosť poľa čísel. Zadanú hodnotu pomocou funkcie *scanf* uložíme do premennej *vstup*. Vzhľadom nato, že veľkosť poľa musí byť kladné celé číslo, musíme tento vstup ešte ošetriť tak, že tieto príkazy vložíme do slučky „*do while*“, ktorá sa bude vyzývať používateľa k zadaniu veľkosti poľa a bude sa opakovať, až dokým zadaná veľkosť nebude kladné celé číslo. (vlastné spracovanie)

Nasledovná časť kódu je implementáciou tohto postupu:

```
do
{
	printf("Zadaj velkost pola cisel: ");
	scanf("%d", &vstup);
	if (vstup < 0)
		printf("Minimalna velkost pola je 1. \n");
	if ((vstup % 1) != 0)
	printf("Velkost pola musi byt kladne cele cislo. \n");
} while ((vstup < 0) || ((vstup % 1) != 0));
```

4.2 Vytvorenie dynamického poľa

Teraz prejdeme k vytvoreniu poľa samotného. Keďže pred spustením programu nevieme, aká bude veľkosť poľa, použijeme dynamické pole, ktorého veľkosť je možné vybrať v priebehu chodu programu. Vytvoríme ho pomocou funkcie *malloc*, ktorá je definovaná v knižnici *stdlib.h*, preto túto knižnicu vložíme do zdrojového kódu direktívou preprocesora „*#include <stdlib.h>*“. Funkcia *malloc* alokuje špecifické množstvo pamäte, ktorú budeme potrebovať na uloženie celého dynamického poľa a vráti ukazovateľ na túto alokovanú pamäť, aby sme k nej nestratili prístup. (Zdroj: ISO/IEC 9899:2011, s. 349)

Na našom počítači je veľkosť dátového typu *long int* 4 bajty. To znamená, že ak bude pole obsahovať *N* prvkov, množstvo alokovanej pamäte bude *N* * 4 bajty.

Pole bude mať názov *dyn_pole* a pamäť preňho alokujeme nasledujúcim príkazom: „*long int* dyn_pole = (long int*)malloc(vstup * sizeof(long int));*“. Operátor *sizeof* vypočíta veľkosť svojho operandu v bajtoch. (Zdroj: ISO/IEC 9899:2011, s. 90)

Vzhľadom nato, že máme dva triediace algoritmy a s každým budeme pole zoraďovať aj vzostupne aj zostupne, museli by sme po každom zoradení poľa vrátiť pole do pôvodného, neusporiadaného stavu. Jednoduchším riešením pre nás bude alokovať pamäť pre ďalšie tri polia, ktoré naplníme na pred usporadúvaním rovnakými číslami v rovnakom poradí, ako prvé pole a tým nám vzniknú štyri identické polia. Každé z týchto polí sa potom bude usporadúvať iným spôsobom. (vlastné spracovanie)

Alokovanie pamäte pre dynamické polia uskutočníme nasledujúcou časťou kódu:

```
long int* dyn_pole = (long int*)malloc(vstup * sizeof(long int));
long int* kopia_pola = (long int*)malloc(vstup * sizeof(long int));
long int* kopia_pola1 = (long int*)malloc(vstup * sizeof(long int));
long int* kopia_pola2 = (long int*)malloc(vstup * sizeof(long int));
```

4.3 Uvoľnenie pamäte

Pri dynamickom alokovaní pamäte musíme myslieť nato, že nepotrebnú alokovanú pamäť treba uvoľniť, aby ju operačný systém mohol prideliť iným aplikáciám. Keďže na konci nášho programu už viac nebudeme potrebovať naše dynamické polia, pamäť potrebnú na ich uloženie môžeme uvoľniť. Toto uvoľnenie vykonáme pomocou funkcie *free*, ktorej ako argument vložíme ukazovateľ na alokovanú pamäť, ktorá bude prichystaná na ďalšie pridelenie. Funkcie *free* pridáme na koniec tela funckie *main*, pred príkaz „*return 0;*“. (vlastné spracovanie)

```
	free(dyn_pole);
	free(kopia_pola);
	free(kopia_pola1);
	free(kopia_pola2);
```

4.4 Naplnenie dynamických polí celými číslami

Po vytvorení dynamických polí musíme tieto polia naplniť číslami. V zdrojovom súbore *funkcie.c* vytvoríme funkciu s názvom *naplnPole*. Jej deklaráciu pridáme do hlavičkového súboru *deklaracie.h*. Táto funkcia bude mať dva parametre, ukazovateľ na dynamické pole a počet prvkov poľa. Do tejto funkcie vložíme generátor pseudonáhodných celých čísel, ktorý upravíme podľa našich požiadaviek. Vybrané triediace algoritmy chceme otestovať pri usporadúvaní polí s veľkým množstvom prvkov ale zároveň aj chceme, aby hodnoty prvkov v poli boli čo najväčšie. Naše majú prvky dátového typu *long int*. Tento dátový typ nám umožňuje uložiť do poľa prvky s hodnotami vyše dvoch miliárd. Generátor pseudonáhodných čísel teda nastavíme tak, aby generoval hodnoty až do 2 100 000 000 a vložíme ho do slučky *do while,* ktorá bude tento generátor spúšťať dovtedy, kým vygenerované číslo nebude väčšie ako 1 000 000 000. Funkcia *naplnPole* pomocou slučky *for* naplní pole číslami tak, že vytvorí index *i*, ktorý najskôr nadstaví na hodnotu 0. Generátor pseudonáhodných čísel vygeneruje číslo, ktoré vloží do prvku *dyn_pole[i]*. Po vykonaní všetkých príkazových riadkov v cykle *for* sa hodnota indexu *i* zvýši o 1 a pokračuje sa ďalšími iteráciami dovtedy, dokým bude index *i* menší ako počet prvkov. Inými slovami, až kým sa nenaplní číslami celé pole. (vlastné spracovanie)

Kód tejto funkcie v jazyku C:

```
void naplnPole(long int* dyn_pole, long int vstup)
{
	for (long int i = 0; i < vstup; i++)
	{
		long int a;
		do
		{
			a = (double)rand() / (RAND_MAX + 1) * 2100000000 + 1;
		} while (a < 1000000000);
		dyn_pole[i] = a;
	}
}
```

4.5 Generovanie pseudonáhodných čísel

Generovanie pseudonáhodných čísel nám zabezpečuje funkcia *rand*, ktorá je definovaná v knižnici *stdlib.h*. Táto funkcia vracia sekvenciu na prvý pohľad nezávislých celých čísel v rozsahu 0 až *RAND_MAX* zakaždým, keď je zavolaná. Konštanta *RAND_MAX* predstavuje maximálnu hodnotu, ktorú môže funkcia *rand* vrátiť a je zaručené, že jej hodnota je minimálne 32 767. (Zdroj: ISO/IEC 9899:2011, s. 346)

Na generovanie určitej série čísel používa táto funkcia začiatočnú hodnotu nazývanú *seed*, ktorú pomocou rôznych matematických operácii pretransformuje na inú hodnotu tak, aby sa javila ako nezávislá na hodnote, ktorú má *seed*. *Seed* sa inicializuje na určitú hodnotu prostredníctvom funkcie *srand*. Funkcia *srand* používa ako parameter *seed* pre novú sekvenciu pseudonáhodných čísel, ktoré sa majú vrátiť následným volaním *rand*. Ak sa *srand* potom znovu zavolá s rovnakou hodnotou *seedu*, sekvencia pseudonáhodných čísel bude rovnaká ako pri prvom zavolaní. Ak je volanie funkcie *rand* uskutočnené predtým, ako bola zavolaná funkcia *srand*, vygeneruje sa taká postupnosť čísel, ako keby bola zavolaná funkcia *srand* s parametrom s hodnotou 1.
(Zdroj: ISO/IEC 9899:2011, s. 346)

Aby sa teda po každom volaní funkcie *rand* vygenerovali iné čísla, potrebujeme, aby bola hodnota *seedu* zakaždým iná. To dosiahneme tak, že do parametra funkcie *srand* vložíme volanie funkcie *time(NULL)*, ktorá vráti čas, ktorý uplynul od 1. januára 1970 v sekundách. (Bansal, 2018) *Seed* bude mať preto po každom spustení programu inú hodnotu a funkcia *rand* bude zakaždým generovať inú sekvenciu pseudonáhodných čísel. Funkcia *time* je definovaná v knižnici *time.h*, ktorú tiež pridáme do zdrojového kódu.

Ak chceme, aby nám funkcia *rand* generovala celé čísla od 1 po 2 100 000 000, do zdrojového kódu ju implementujeme nasledujúcim príkazom:

```
a = (double)rand() / (RAND_MAX + 1) * 2100000000 + 1;
```

Teraz príkazom „*srand(time(NULL));*“ nastavíme *seed*. Príkaz ideálne umiestnime hneď na začiatok funkcie *main*, aby sa hodnota *seedu* nastavila hneď po spustení programu.

Príkaz „*naplnPole(dyn_pole, vstup);*“ zavolá funkciu *naplnPole*, ktorá naplní pole pseudonáhodnými číslami od 1 000 000 000 po 2 100 000 000.

Naplnenie ostatných kópií poľa už neuskutočníme funkciou *naplnPole*, pretože by sme v každom poli mali iné čísla, čo by mohlo skresliť výsledky exekučných časov. Ostatné polia naplníme jednoduchými cyklami *for*, v ktorých sa nastaví počiatočná hodnota indexu *i* na 0. Hodnota prvku *dyn_pole[i]* sa skopíruje do prvku *kopia_pola[i]* a index *i* sa zvýši

o 1. Tento proces sa opakuje dovtedy, dokým bude *i* menšie ako počet prvkov v poli. (vlastné spracovanie)

Volanie funkcie naplnPole a naplnenie kópií poľa sa zrealizuje nasledujúcou časťou kódu:

```
naplnPole(dyn_pole, vstup);
for
     (long int i = 0; i < vstup; i++)
     kopia_pola[i] = dyn_pole[i];
for
     (long int i = 0; i < vstup; i++)
     kopia_pola1[i] = dyn_pole[i];
for
     (long int i = 0; i < vstup; i++)
     kopia_pola2[i] = dyn_pole[i];
```

4.6 Implementácia triediacich algoritmov do programu

Polia máme pripravené, teraz sa môžeme venovať implementácii algoritmov. Zdrojové kódy algoritmov Shaker sort a Shell sort, ktoré sme spomenuli a ich fungovanie sme opísali v prechádzajúcich kapitolách môžeme s drobnou úpravou spolu s definovaným makrom „*vymenCisla*“ pridať do zdrojového súboru *funkcie.c*. Keďže potrebujeme tieto funkcie rozlíšiť od tých, ktoré nám budú pole usporadúvať zostupne, nazveme tieto funkcie „*ShakeSortVzostupne*“ a „*ShellSortVzostupne*“. Tiež musíme zmeniť definíciu ich prvého parametra zo statického poľa celých čísel na ukazovateľ na dynamické celočíselné pole.
Úpravami týchto funkcií vytvoríme dve ďalšie funkcie „*ShakeSortZostupne*“ a „*ShellSortZostupne*“. (vlastné spracovanie)

Zdrojové kódy týchto funkcií sú:

```
void ShakeSortZostupne(long int *dyn_pole, long int vstup)
{

	long int lavy = 1, pravy = vstup - 1, i, k;
	do {
		for (i = pravy; i >= lavy; i--)
			if (dyn_pole[i - 1] < dyn_pole[i])
			{
				vymenCisla(dyn_pole[i - 1], dyn_pole[i]); k = i;
			}
		lavy = k + 1;
		for (i = lavy; i <= pravy; i++)
			if (dyn_pole[i - 1] < dyn_pole[i])
			{
				vymenCisla(dyn_pole[i - 1], dyn_pole[i]); k = i;
			} pravy = k - 1;
	} while (lavy < pravy);

}
```

```
void ShellSortZostupne(long int *dyn_pole, long int vstup)
{
	long int m = vstup; while (m /= 2)
		for (long int d = vstup - m, i = 0; i < d; i++)
			for (long int j = i; (j >= 0) && (dyn_pole[j] < dyn_pole[j +
m]); j -= m)
				vymenCisla(dyn_pole[j], dyn_pole[j + m]);
}
```

V obidvoch prípadoch nám stačilo len vymeniť porovnávacie znamienka pri porovnávaní dvoch prvkov tak, aby funkcia vymenila ich pozície vtedy, keď je menší prvok napravo od väčšieho.

Deklarácie všetkých týchto funkcií pridáme do hlavičkového súboru *deklaracie.h*. (vlastné spracovanie)

4.7 Testovacie funkcie

Ešte pred porovnávaním exekučných časov chceme mať istotu, že sa polia naozaj usporiadali a sú zoradené správne. Niekedy sa môže napríklad stať, že funkcia *malloc* nepridelí žiadnu pamäť pre naše dynamické polia, pretože operačný systém nemusí mať žiadnu pamäť na vybavenie žiadosti o pridelenie a tým sa usporadúvajúce funkcie nespustia správne. Aby sme sa teda ubezpečili, že sa pri chode programu nestala žiadna chyba a polia sú usporiadané, vytvoríme si testovacie funkcie, ktoré fungujú na jednoduchom princípe. Funkcie postupne prechádza pole zľava doprava, porovnáva susediace prvky a kontroluje, či je prvok vľavo menší (ak ide o funkciu, ktorá testuje zostupnosť, tak prvok musí byť väčší)

ako susediaci prvok napravo. Pri úspešnom teste program vypíše na konzolu, že test prebehol úspešne. Funkcie nazveme „*TestVzostupne*“ a „*TestZostupne*“. Tak ako každú nami vytvorenú funkciu, aj tieto funkcie vložíme do zdrojového súboru *funkcie.c* a ich deklarácie vložíme do hlavičkového súboru *deklaracie.h*. (vlastné spracovanie)
Zdrojové kódy funkcií *TestVzostpune* a *TestZostupne*:

```
int TestVzostupne(long int *dyn_pole, long int vstup)
{
	for (long int i = 1; i < vstup; i++)
		if (dyn_pole[i - 1] > dyn_pole[i])
		return 0;
	return 1;
}
```

```
int TestZostupne(long int *dyn_pole, long int vstup)
{
	for (long int i = 1; i < vstup; i++)
		if (dyn_pole[i - 1] < dyn_pole[i])
			return 0;
		return 1;
}
```

4.8 Minimálny a maximálny prvok v poli

Vzhľadom nato, že čísla v poli budú vygenerované generátorom pseudonáhodných čísel, chceme mať aspoň prehľad o tom, v akom rozsahu boli čísla vygenerované. Nato použijeme funkcie „*minCislo*“ a „*maxCislo*“, ktoré vrátia hodnotu indexov najmenšieho a najväčšieho prvku v poli. Do ich parametrov vložíme ukazovateľ na dynamické pole a počet prvkov v poli. Funkcie vložíme do súboru *funkcie.c* a ich deklarácie do hlavičkového súboru *deklaracie.h*. (vlastné spracovanie)

Zdrojový kód funkcie *minCislo*:

```
long int minCislo(long int* dyn_pole, long int vstup)
{
	int c, min, index;

	min = dyn_pole[0];
	index = 0;

	for (c = 1; c < vstup; c++)
	{
		if (dyn_pole[c] < min)
		{
			index = c;
			min = dyn_pole[c];
		}
	}
	return index;
}
```

Funkcia *minCislo* vloží do premennej *min* hodnotu prvého prvku v poli *dyn_pole[0]*. Slučka *for* postupne prechádza poľom a porovnáva ostatné prvky poľa s premennou *min*. Ak nájde prvok menší ako *min*, hodnotu min prepíše na hodnotu menšieho prvku a index tohto prvku uloží do premennej *index*. Po ukončení cyklu vráti funkcia premennú *index*, v ktorej je uložený index najmenšieho prvku v poli. (vlastné spracovanie)

Zdrojový kód funkcie *maxCislo*:

```
long int maxCislo(long int* dyn_pole, long int vstup)
{
	int c, max, index;

	max = dyn_pole[0];
	index = 0;

	for (c = 1; c < vstup; c++)
	{
		if (dyn_pole[c] > max)
		{
			index = c;
			max = dyn_pole[c];
		}
	}
	return index;
}
```

Na rovnakom princípe funguje aj funkcia *maxCislo*. Pomocou cyklu *for* prechádza poľom a hľadá najväčší prvok, ktorého index uloží do premennej *index* a po skončení funkcie vráti jeho hodnotu.

Vďaka týmto dvom indexom teraz vieme, na ktorých pozíciách v poli sa nachádza najmenší a najväčší prvok.

V tele funkcie *main* si definujeme štyri nové premenné dátového typu *long int* s názvami: *poloha*, *poloha1*, *minimum*, *maximum*. Do premenných *poloha* a *poloha1* uložíme indexy vrátené funkciami *minCislo* a *maxCislo*. Do premennej *minimum* potom vložíme hodnotu prvku *dyn_pole[poloha]* a do premennej *maximum* hodnotu prvku *dyn_pole[poloha1]*. Tieto dve premenné teraz držia hodnotu minimálneho a maximálneho prvku v poli.
(vlastné spracovanie)
Ich výpis na konzolu uskutočníme nasledujúcou časťou kódu:

```
poloha = minCislo(dyn_pole, vstup);
minimum = dyn_pole[poloha];
poloha1 = maxCislo(dyn_pole, vstup);
maximum = dyn_pole[poloha1];
printf("Minimalny prvok v poli: ");
printf("%d \n", minimum);
printf("Maximalny prvok v poli: ");
printf("%d \n\n", maximum);
```

4.9 Meranie exekučného času

Jednou z najdôležitejších častí pre porovnávanie algoritmov je samotné zmeranie dĺžky exekučného času triediacich algoritmov implementovaných do funkcií. Na toto meranie nám poslúži funkcia *clock* definovaná v hlavičkovom súbore *time.h*. Táto funkcia vracia počet takzvaných „tikov“ od začiatku spustenia programu. Pre vyjadrenie času v sekundách musíme vrátenú hodnotu vydeliť konštantou „*CLOCKS_PER_SEC*“.
(Zdroj: ISO/IEC 9899:2011, s. 386)

Funkcia *main* zavolá funkciu *clock* tesne pred zavolaním zoraďovacej funkcie a tesne po jej zavolaní. Obidva tieto časy uložíme do premenných *c1* a *c2* dátového typu *clock_t*, ktorý dokáže reprezentovať počet „tikov“. Zmerané časy od seba odpočítame, výsledok vydelíme konštantou „*CLOCKS_PER_SEC*“ a dostaneme exekučný čas zoraďovacej funkcie v sekundách. Tento výsledok program vypíše na konzolu a zapíše do externého súboru.
(vlastné spracovanie)

4.10 Vytvorenie externého súboru

Aby sme výsledky mohli porovnávať, budeme ich ukladať do programom vytvoreného externého súboru, pretože inak by sme výsledky meraní po ukončení programu a zatvorení konzoly stratili. Na vytvorenie externého súboru potrebujeme deklarovať ukazovateľ na dátový typ „*FILE*“. Deklarácia ukazovateľa je nasledovná: „*FILE* out;*“

Vytvorenie dátového prúdu sa uskutočňuje pomocou funkcie *fopen* definovanej v súbore *stdio.h.* Táto funkcia má dva parametre. Do prvého sa vkladá názov súboru, z ktorého sa má vytvoriť dátový prúd. Ak tento súbor neexistuje, program ho vytvorí.
(vlastné spracovanie)

4.10.1 Názov externého súboru

Do názvu súboru pridáme aj časovú pečiatku, aby bolo zrejmé, kedy bol súbor vytvorený. Musíme najprv inicializovať premennú s názvom „*vysledok*“, ktorá bude dátového typu *char*. Pomocou funkcie *strftime* vložíme do premennej názov externého súboru spolu s časovým formátom, aký nám vyhovuje. (Zdroj: ISO/IEC 9899:2011, s. 391) Premennú *vysledok* vložíme do prvého parametra funkcie *fopen*. (vlastné spracovanie)

Časovú pečiatku do názvu externého súboru vložíme pomocou tohto kódu:

```
time_t rawtime;
struct tm* timeinfo;
char vysledok[50];
time(&rawtime);
timeinfo = localtime(&rawtime);
strftime(vysledok, 50, "Vysledok %d-%m-%Y %H_%M_%S.dat", timeinfo);
```

Druhý parameter určuje mód, v ktorom sa súbor otvorí. Mód napríklad určuje, či sa má súbor otvoriť len na čítanie alebo len na zápis, atď. Ako druhý parameter zvolíme mód „*a*“, čo znamená, že ak už existuje súbor s rovnakým názvom, program nebude prepisovať jeho obsah, ale začne údaje zapisovať na koniec tohto súboru.
(Zdroj: ISO/IEC 9899:2011, s. 305)

Na záver už len pridáme podmienku *if*, ktorá zabezpečí, že ak sa dátový prúd nepodarí vytvoriť, program vypíše na konzolu upozornenie a ukončí sa.

```
      FILE *out;
      long int vstup, poloha, poloha1, minimum, maximum;
      out = fopen(vysledok, "a");
      if (!out)
      {
            printf("Subor sa neda vytvorit !!!");
            return 1;
                                    }
```

4.10.2 Zápis do externého súboru

Keď už máme externý súbor vytvorený, môžeme vytvoriť funkciu pre zápis výsledkov. Funkcia bude mať názov „*vypisSubor*“ a bude mať tri parametre. Prvý bude ukazovateľ na dátový prúd, a ďalšie dva budú premenné *c1* a *c2* v ktorých máme uložené počty tikov pred zavolaním triediacej funkcie a po jej zavolaní. Funkcia vypočíta exekučný čas triediacej funkcie v sekundách a zapíše ho do externého súboru.

Zdrojový kód funkcie *vypisSubor*:

```
void vypisSubor(FILE *vystup, clock_t c1, clock_t c2)
{
      fprintf(vystup, "%f sec\n", ((double)(c2 - c1)) /
CLOCKS_PER_SEC);
      fprintf(vystup, "\n");
}
```

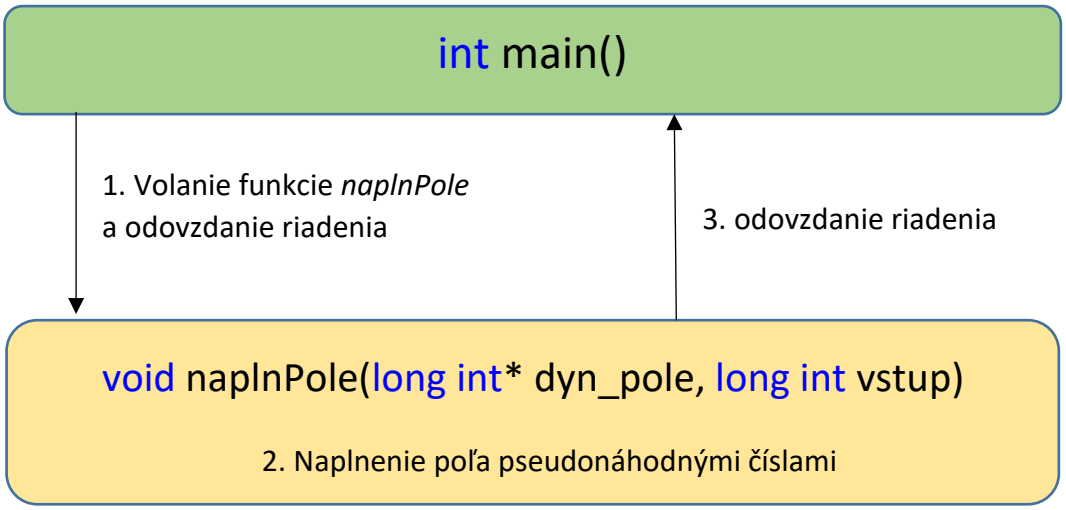
Ostatné informácie, ako je napríklad veľkosť zoraďovaného poľa, minimálny a maximálny prvok v poli alebo dátový typ poľa budeme uskutočňovať pomocou vstavanej funkcie *fprintf*, ktorá slúži na zápis reťazcov, znakov alebo čísel do dátového prúdu.
(vlastné spracovanie)

4.11 Sumárny prehľad vytvorených funkcií

Táto kapitola obsahuje celkový prehľad nami vytvorených funkcií so stručným komentárom a znázornením ich volania funkciou *main* a odovzdávania si návratovej hodnoty a riadenia.

Main je špeciálna funkcia, ktorú musí obsahovať každý program napísaný v jazyku C. Pri spustení programu sa začína prvým riadkom tejto funkcie a pokračuje sa sekvenčne.

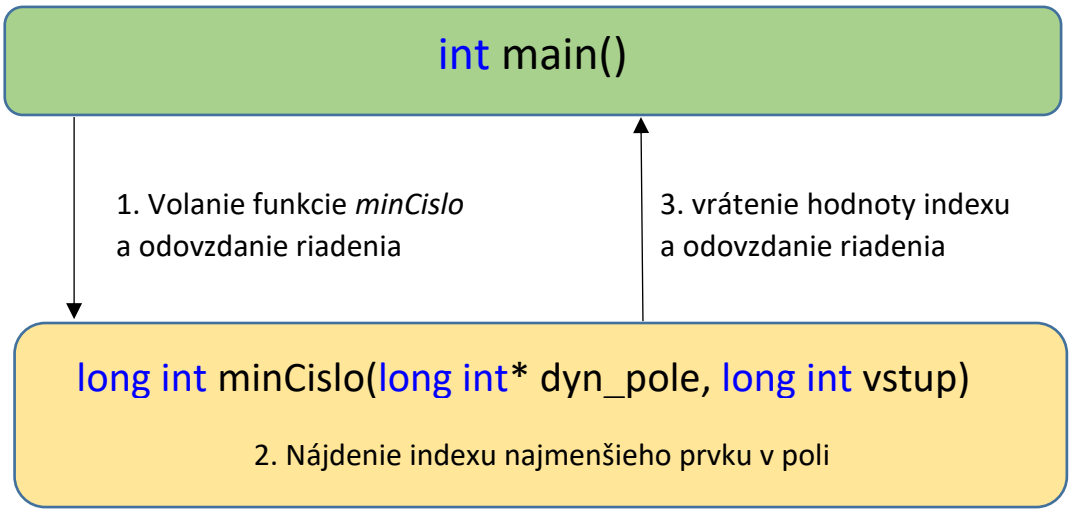
4.11.1 Funkcia *naplnPole*



Obrázok č. 9 – Diagram volania funkcie *naplnPole* funkciou *main* (vlastné spracovanie)

Funkcia *main* zavolá funkciu *naplnPole*, odovzdá jej riadenie a zároveň sa preruší priebeh funkcie *main*. Sekvenčne sa začnú vykonávať jednotlivé príkazy v tele funkcie *naplnPole*, ktorá má za úlohu naplniť dynamické pole pseudonáhodnými číslami. Ukazovateľ na toto dynamické pole sme vložili do prvého parametra. Táto funkcia má návratový typ *void*, takže nevracia funkcii *main* žiadnu hodnotu, iba naplní dynamické pole číslami a po skončení vráti riadenie naspäť funkcii *main*, ktorá pokračuje v bode, v ktorom sa prerušila. (vlastné spracovanie)

4.11.2 Funkcie minCislo a maxCislo

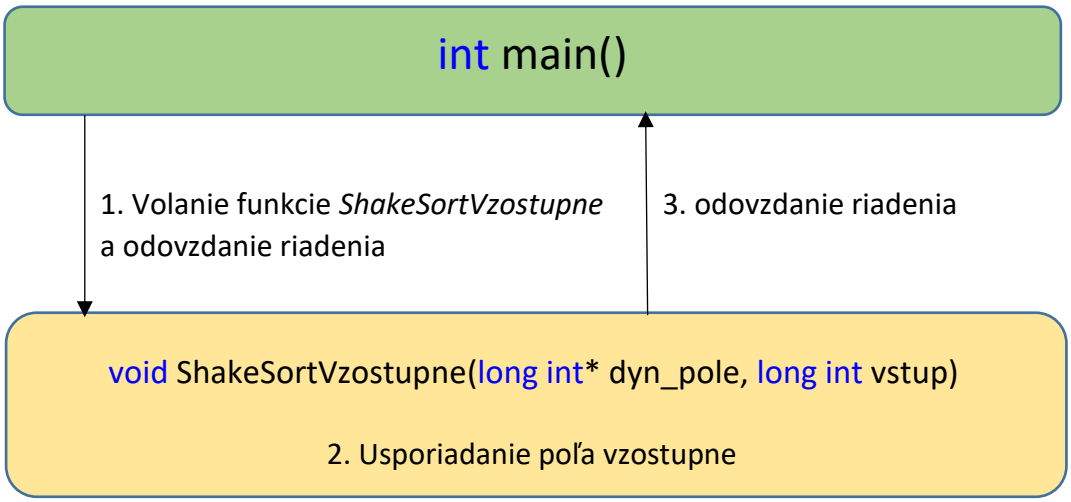


Obrázok č. 10 – Diagram volania funkcie *minCislo* funkciou *main* (vlastné spracovanie)

Po zavolaní funkciou *main* sa funkcia *minCislo* spustí, prevezme riadenie, porovná prvky v poli, ktorého ukazovateľ sme vložili do parametra a nájde index najmenšieho prvku. Po skončení vráti index najmenšieho prvku funkcii *main*, ktorá prvok s týmto indexom vloží do premennej minimum.

Na rovnakom princípe funguje aj volanie funkcie *maxCislo*, ktorá vracia index najväčšieho prvku v poli. (vlastné spracovanie)

4.11.3 Funkcie ShakeSortVzostupne a ShakeSortZostupne

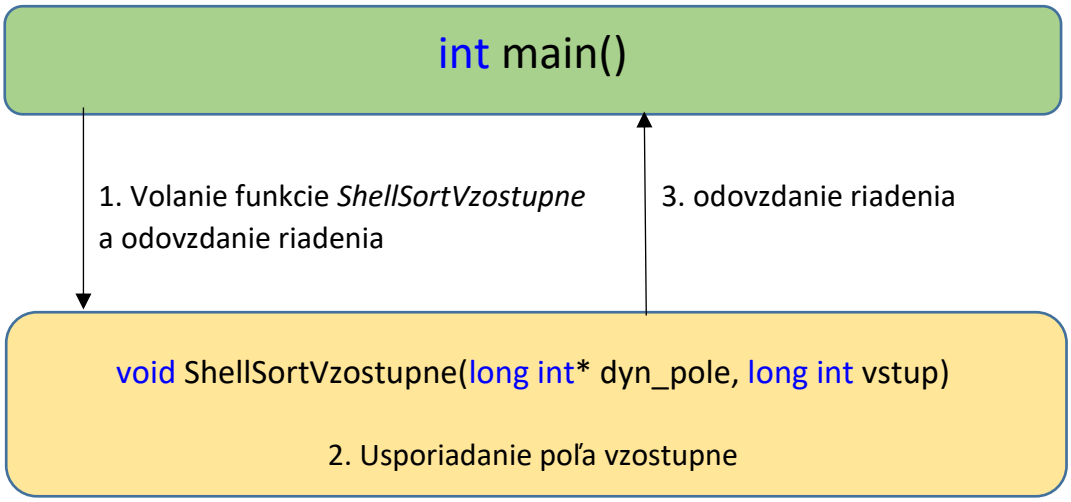


Obrázok č. 11 – Diagram volania funkcie *ShakeSortVzostupne* funkciou *main* (vlastné spracovanie)

Funkcia *main* zavolá funkciu *ShakeSortVzostupne* a prenechá jej riadenie. Po spustení táto funkcia zoradí vzostupne algoritmom Shake sort pole, ktorého ukazovateľ má funkcia vložený do parametra. Táto funkcia má návratový typ *void*, takže funkcii *main* nevracia naspäť žiadnu hodnotu, jej úlohou bolo len usporiadať prvky v poli.

Rovnako funguje aj volanie upravenej verzie tejto funkcie *ShakeSortZostupn*e, ktorá usporadúva prvky od najväčšieho po najmenší. (vlastné spracovanie)

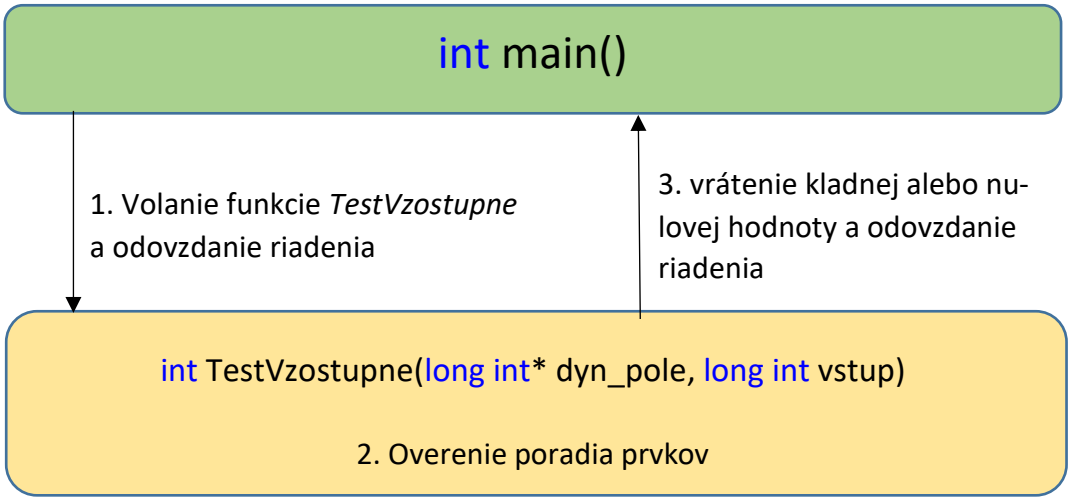
4.11.4 Funkcie ShellSortVzostupne a ShellSortZostupne



Obrázok č. 12 – Diagram volania funkcie *ShellSortVzostupne* funkciou *main* (vlastné spracovanie)

Volanie a odovzdávanie riadenia funkciám *ShellSortVzostupne* a *ShellSortZostupne* prebieha totožne, ako volanie funkcií *ShakeSortVzostupne* a *ShakeSortZostupne*. Tak isto sa jedná o funkcie s návratovým typom *void*, takže nevracajú volajúcej funkcii *main* žiadnu hodnotu, iba zoradia pole algoritmom Shell sort. (vlastné spracovanie)

4.11.5 Funkcie TestVzostupne a TestZostupne



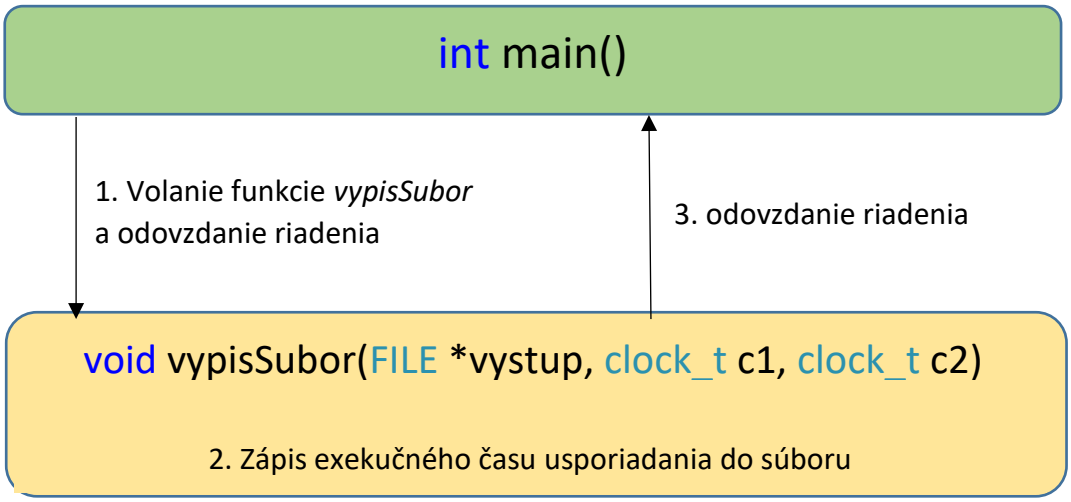
Obrázok č. 13 – Diagram volania funkcie *TestVzostupne* funkciou *main* (vlastné spracovanie)

Po zavolaní funkcie *TestVzostupne* sa funkcia spustí a prevezme riadenie. Overí, či sú prvky v poli naozaj zoradené vzostupne. Táto funkcia má návratový typ *int*, takže vracia volajúcej funkcii celočíselnú hodnotu. V tomto prípade vráti funkcia volajúcej funkcii *main*

buď 0, v prípade, že je pole usporiadané správne alebo hodnotu 1 v prípade, že sa pole správne neusporiadalo. Po vrátení hodnoty sa funkcia ukončí a volajúca funkcia *main* opäť prevezme riadenie. Podľa návratovej hodnoty funkcie *TestVzostupne* vyhodnotí, či má pokračovať v ďalších zoraďovaniach polí alebo či má v prípade, že sa pole neusporiadalo správne, ukončiť program.

Volanie funkcie, odovzdávanie návratovej hodnoty a riadenia pri funkcii *TestZostupne* je rovnaké, ale funkcia overuje, či je pole usporiadané zostupne.

4.11.6 Funkcia *vypisSubor*



Obrázok č. 14 – Diagram volania funkcie *vypisSubor* funkciou *main*

(vlastné spracovanie)

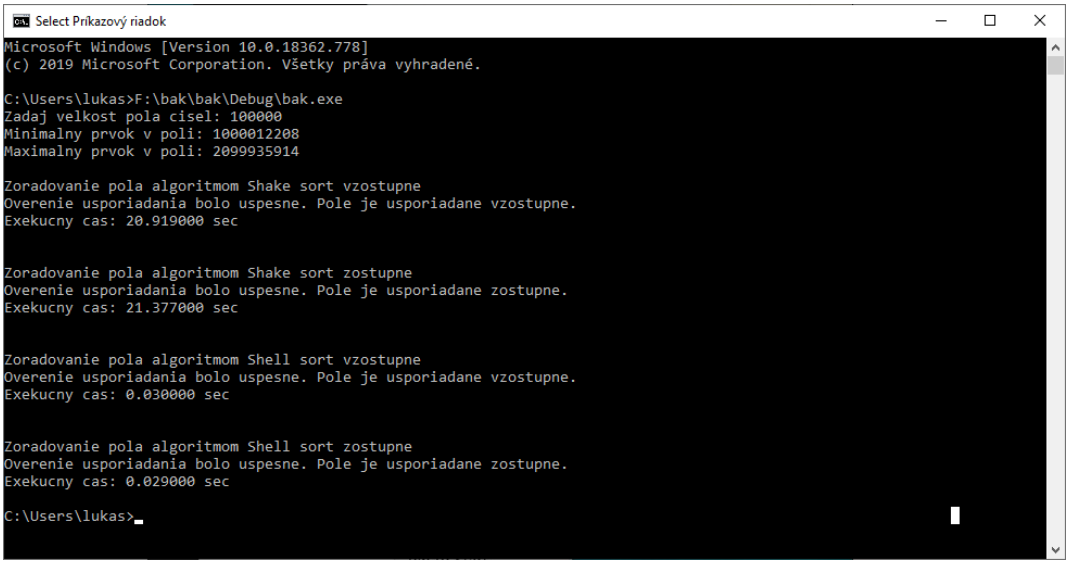
Funkcia *main* zavolá funkciu *vypisSubor*, táto funkcia sa spustí a prevezme riadenie. Funkcia od seba odpočíta časy, ktoré má vložené v druhom a treťom parametri, tým vypočíta exekučný čas zoraďovania, ktorý v sekundách zapíše do dátového prúdu. Ukazovateľ na tento dátový prúd má vložený do prvého parametra. Po skončení znovu prevezme riadenie funkcia *main*.

4.12 Spustenie programu

Všetky funkcie a príkazy potrebné na spustenie programu a usporiadanie poľa máme v našom projekte vytvorené, takže teraz môžeme program skompilovať a spustiť program. Vo vývojom prostredí Visual Studio 2019 vyberieme z horného menu kartu *Build*. Z rolovacieho zoznamu vyberieme možnosť „*Build Solution*“. Ak kompilácia prebehla úspešne, môžeme prejsť k spusteniu programu. Po skompilovaní projektu by sa nám mal

v projektovom adresári vytvoriť spustiteľný súbor s príponou *exe*. Keďže sa jedná o konzolovú aplikáciu, spustíme tento program cez príkazový riadok. V operačnom systéme Windows 10 spustíme program cez príkazový riadok nasledovne:

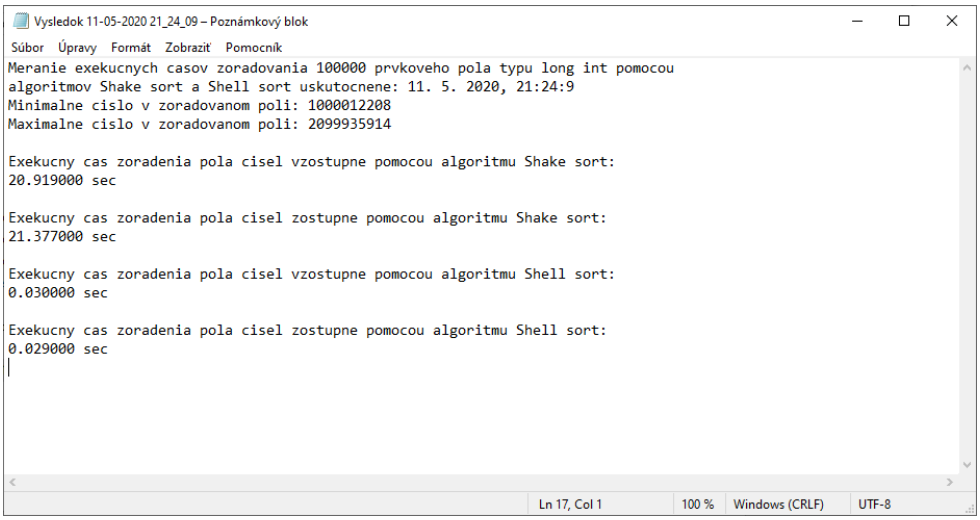
Na klávesnici stlačíme klávesu Windows, ktorou otvoríme menu Štart, zadáme výraz „*cmd*“ a stlačíme klávesu Enter. Otvorí sa nám príkazový riadok, do ktorého zadáme cestu k nášmu spustiteľnému súboru. V našom prípade je to cesta: F:\bak\bak\Debug\bak.exe. (vlastné spracovanie)



Obrázok č. 15 – Program bak.exe spustený v Príkazovom riadku (vlastné spracovanie)

Do prvého riadku programu v sme zadali veľkosť poľa 100 000 prvkov. Program vytvorí dynamické pole tejto veľkosti. Do druhého a tretieho riadku program vypísal na konzolu premenné minimum a maximum, v ktorých je uložený najmenší a najväčší vygenerovaný prvok vo vytvorenom poli. Najmenší prvok v tomto poli je 1 000 012 208 a najväčší prvok je 2 099 935 914. Potom program začal pole usporadúvať, najprv algoritmom Shake sort vzostupne. Pomocou testovacej funkcie *TestVzostupne* overil, že sa program usporiadal úspešne a vypísal exekučný čas tohto zoradenia. Čas potrebný na zoradenie poľa algoritmom Shake sort vzostupne bol v tomto prípade 20,919 sekúnd. Následne vykonal aj ostatné tri spôsoby usporiadania poľa. Vo všetkých prípadoch prebehol test správnosti usporiadania úspešne, program vypísal exekučné časy usporiadaní a skončil.

Program počas svojho chodu tiež vytvoril v projektovom adresári externý súbor, do ktorého zapísal svoje výstupy.



Obrázok č. 16 – Externý súbor so zapísanými výsledkami (vlastné spracovanie)

5 Meranie a porovnanie exekučných časov algoritmov Shake sort a Shell sort

Po vytvorení programu, do ktorého sme implementovali triediace algoritmy môžeme prejsť k porovnávaniu ich exekučných časov. Našou úlohou bude zistiť, ktorý z týchto algoritmov je efektívnejší pri usporadúvaní celočíselných polí s niekoľko stotisíc prvkami. Pri meraní časov nebudeme mať spustenú žiadnu inú aplikáciu, aby bol procesor čo najmenej zaťažený a výsledky čo najmenej skreslené. (vlastné spracovanie)

Triediace algoritmy budeme v programe testovať s piatimi vybranými vstupmi. Veľkosti polí, na ktorých budeme porovnávať algoritmy Shake sort a Shell sort budú: 10 000, 50 000, 100 000, 200 000 a 300 000 prvkov. Pre každú vybranú veľkosť poľa spustíme program pätnásť krát. Pri jednom spustení programu sa zmerajú exekučné časy zoraďovaní vzostupne aj zostupne obi dvoma algoritmami v sekundách. Z výsledkov každého spôsobu usporadúvania spravíme aritmetický priemer, priemery zapíšeme do tabuľky, porovnáme a zistíme, ktorý z vybraných algoritmov je pre zoraďovanie veľkého množstva prvkov efektívnejší. Okrem toho porovnáme aj časy zoraďovania polí vzostupne a zostupne a zistíme, či má prehodenie poradia, akým sa má pole usporiadať vplyv na čas vykonávania usporadúvania. (vlastné spracovanie)

5.1 Predpoklad

Obi dva vybrané triediace algoritmy majú zložitosť $O(N^2)$. S nárastom prvkov v poli bude teda narastať aj exekučný čas algoritmov. Predpokladáme, že algoritmus Shell sort

bude pri zoraďovaní väčších polí efektívnejší, pretože ako sme spomínali v predchádzajúcich kapitolách, z algoritmov, ktoré majú zložitosť $O(N^2)$ patrí k najvýkonnejším, pretože neporovnáva susedné prvky ale aj prvky od seba vzdialené, čím dokáže znížiť počet potrebných porovnaní. (vlastné spracovanie)

5.2 Merania exekučných časov

Tabuľka č. 1 zobrazuje spriemerované výsledky uskutočnených meraní v sekundách.

Algoritmus / Počet prvkov	Shake sort		Shell sort	
	vzostupne	zostupne	vzostupne	zostupne
10000	0,203	0,204	0,002	0,002
50000	5,081	5,054	0,013	0,013
100000	20,350	20,193	0,029	0,030
200000	81,368	81,359	0,065	0,066
300000	186,131	185,417	0,098	0,100

Tabuľka č. 1 – Priemer výsledkov exekučných časov (vlastné spracovanie)

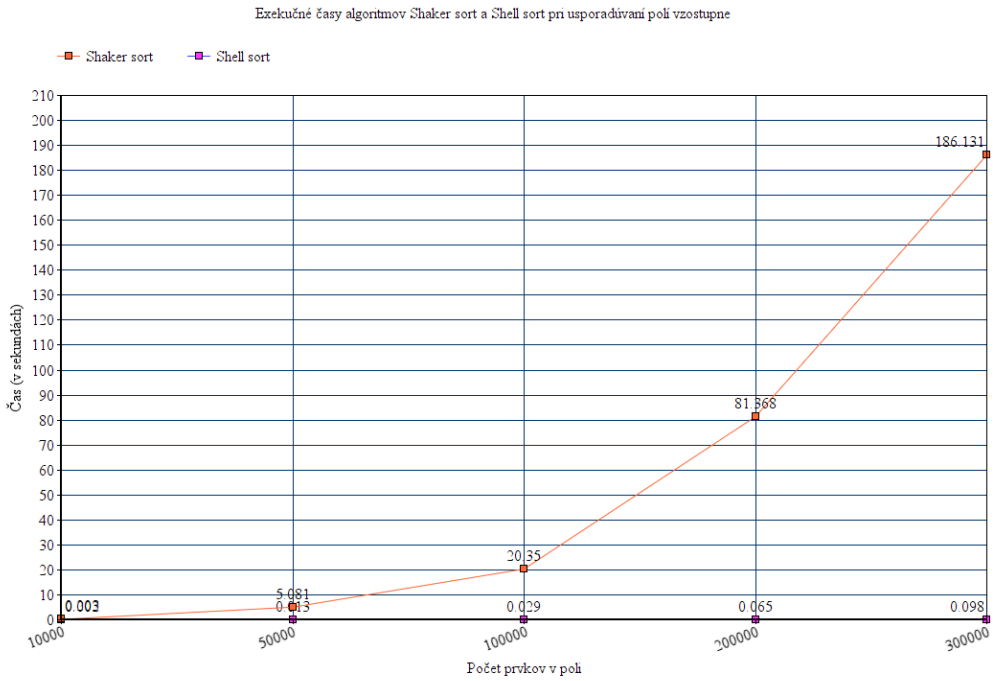
Výsledky meraní dopadli podľa očakávaní. Keďže Shake sort má časovú zložitosť $O(N^2)$, takže ako môžeme vidieť v tabuľke, vždy, keď sa zdvojnásobil počet prvkov v poli, exekučný čas algoritmu narástol približne štvornásobne. Aj keď sa algoritmus Shell sort zaraďuje medzi algoritmy s časovou zložitosťou $O(N^2)$, táto časová zložitosť platí len pre najhoršie možné poradie vstupov a teda najhoršiu dobu chodu algoritmu. Ako môžeme vidieť, pri daných vstupoch a pri zvolenej počiatočnej medzere medzi porovnávanými prvkami sa algoritmus správa ako pri časovej zložitosti O(N log N). Vždy, keď sa prvky v poli zdvojnásobia, exekučný čas algoritmu sa viac ako zdvojnásobí (nie o veľa). (vlastné spracovanie)

5.3 Záver meraní

Ako môžeme vidieť v tabuľke č. 1, algoritmus Shell sort bol pri všetkých zadaných vstupoch rýchlejší ako algoritmus Shake sort. Pri poli s 10 000 prvkami boli časy oboch pomerne uspokojujúce, pri zvyšovaní prvkov sa začalo výrazne ukazovať, ktorý z dvoch porovnávaných algoritmov je efektívnejší. Zatiaľ čo exekučný čas Shake sortu s každým

zvýšením počtu prvkov narástol o niekoľko sekúnd, Shell sort zvládol každé pole usporiadať do jednej desatiny sekundy.

Na obrázku č. 17 je graf, ktorý porovnáva exekučné časy oboch algoritmov pri rôznych veľkostiach polí pri usporadúvaní vzostupne. (vlastné spracovanie)



Obrázok č. 17 – Exekučné časy algoritmov Shake sort a Shell sort pri usporadúvaní polí vzostupne (vlastné spracovanie)

Pri 300 000 prvkovom poli bol časový rozdiel medzi Shake sortom a Shell sortom približne až 186 sekúnd, čo sú v prepočte približne 3 minúty. Znamená to teda, že ak by sme chceli napríklad vytvoriť program na tvorbu databáz, ktorá by mohla obsahovať niekoľko stotisíc záznamov, bolo by nevhodné zvoliť triediaci algoritmus Shake sort, pretože používateľ programu by musel na každé usporiadanie čakať niekoľko minút. Naopak, algoritmus Shell sort by bol pre aplikovanie do praktického programu, v ktorom očakávame, že sa budú triediť veľké množstvá dát veľmi vhodný, pretože zoraďovanie veľkých polí zvládol za veľmi krátky čas. (vlastné spracovanie)

Porovnaním spriemerovaných časov sme tiež zistili, že to, či sa pole zoraďuje vzostupne alebo zostupne nemá žiadny vplyv na exekučný čas algoritmu, v oboch prípadoch boli časy usporadúvania takmer totožné. (vlastné spracovanie)

Záver

Implementáciou triediacich algoritmov Shake sort a Shell sort do programu v jazyku C a porovnaním viacerých výsledkov z meraní exekučných časov sa nám úspešne podarilo dokázať, že pri usporadúvaní veľkých súborov dát, ako je napríklad pole s niekoľko stotisíc prvkami, pracuje algoritmus Shell sort výrazne rýchlejšie a efektívnejšie.

Zoznam použitej literatúry

Knihy a monografie

SEDGEWICK, R. *Algorithms in C*. 2. vyd. Boston: Addison-Wesley, 1990. 657 s. ISBN 0-201-51425-7.

CORMEN, T. et al. *Introduction to Algorithms*. 3. vyd. Cambridge: The MIT Press, 2009. 1 292 s. ISBN 978-0-262-53305-8.

SKIENA, S. *The Algorithm Design Manual*. 2. vyd. Londýn: Springer Science & Business Media, 2009. 730 s. ISBN 978-1-84800-069-8.

MILLER, B., RANUM, D. *Problem Solving with Algorithms and Data Structures Using Python*. Portland: Franklin, Beedle & Associates, 2011. 432 s. ISBN 978-1590282571.

KNUTH, D. *The Art of Computer Programming: Volume 3 / Sorting and Searching*, 2. vyd. Boston: Addison-Wesley, 1998. 782 s. ISBN 978-0201896855.

Články v elektronických časopisoch a iné príspevky

HORDĚJČUK, V. *Asymptotická složitost*. In: voho.eu. Luxemburg, [2008], online. [cit. 2020-04-27] Dostupné na: http://voho.eu/wiki/asymptoticka-slozitost/

SAHAI, H. Cocktail Shaker Sort / Bidirectional bubble sort. In: iq.opengenus.org. Dillí, [2019], online. [cit. 2020-04-27] Dostupné na: https://iq.opengenus.org/cocktail-shaker-sort/

NAYAL, CH. et al. *Insertion sort*. In: geegsforgeeks.org. Dehradun, [2013] online. [cit. 2020-05-01] Dostupné na: https://www.geeksforgeeks.org/insertion-sort/

NAYAL, CH. et al. *Bubble sort*. In: geegsforgeeks.org. Dehradun, [2014] online. [cit. 2020-05-01] Dostupné na: https://www.geeksforgeeks.org/bubble-sort/

Normy

ISO/IEC 9899:2011 : Information technology - Programming languages - C