Žilinská univerzita v Žiline
Fakulta elektrotechniky a informačných Technológií
Katedra mechatroniky a elektroniky

28260620201114

BAKALÁRSKA PRÁCA

Algoritmizácia úloh na mikroprocesorovej platforme MSP430

Jún 2020 **Vincent Uhliarik**

ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA ELEKTROTECHNIKY A INFORMAČNÝCH TECHNOLÓGIÍ
KATEDRA MECHATRONIKY A ELEKTRONIKY

Algoritmizácia úloh na mikroprocesorovej platforme MSP430

BAKALÁRSKA PRÁCA

Študijný program:	Autotronika
Študijný odbor:	5.2.9 Elektrotechnika
Školiace pracovisko:	Žilinská univerzita v Žiline, Fakulta elektrotechniky a informačných Technológií, Katedra mechatroniky a elektroniky
Školiteľ:	Doc. RNDr. Juraj Pančík, CSc.
Konzultant:	

Jún 2020 **Vincent Uhliarik**

ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA ELEKTROTECHNIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

KATEDRA MECHATRONIKY A ELEKTRONIKY

Univerzitná 8215/1, 010 26 Žilina
tel.: 041/5131601 fax:041/5131524 e-mail: kme@fel.uniza.sk

ZADANIE BAKALÁRSKEJ PRÁCE

Meno: **Vincent UHLIARIK**

Študijný program: Autotronika

Názov bakalárskej práce:

Algoritmizácia úloh na mikroprocesorovej platforme MSP430

Pokyny pre vypracovanie bakalárskej práce:

1. Teoretické vymedzenie pojmov (algoritmus, spôsoby zápisu algoritmov, zápis algoritmov pomocou vývojových diagramov, výpočtová zložitosť, triedenie a triediaci algoritmus)
2. Prehľad vybraných algoritmov so zameraním na triediace a numerické algoritmy
3. Návrh a realizácia vybraných algoritmov na mikroprocesorovej platforme MSP430

Predpokladaný rozsah práce - počet strán textu: max. 40
počet strán grafických príloh: max. 10

Školiteľ bakalárskej práce: doc. RNDr. Juraj Pančík, CSc.

Konzultant bakalárskej práce :

Dátum zadania bakalárskej práce: **31. 10. 2019**

Dátum odovzdania bakalárskej práce: **12. 5. 2020**

doc. Ing. Michal Frivaldský, PhD.
vedúci katedry

Abstrakt

Bakalárska práca sa zaoberá návrhom praktických úloh s implementáciou na mikroprocesorovú platformu. Práca je rozdelená do troch hlavných častí. V prvej teoretickej časti sa práca venuje definícií základných termínov. V druhej časti sa práca venuje návrhu riešení numerických a triediacich algoritmov pomocou pseudokódov, vývojových diagramov, tabuliek, jazyka C. V poslednej realizačnej časti sú jednotlivé riešenia algoritmov zhrnuté do tabuľky a implementované na platformu MSP430.

Kľúčové slová: Algoritmus, Vývojový diagram, Triedenie, Pseudokód

Abstract

Bachelor thesis deals with a design of practical tasks with implementation on a microprocessor platform. The thesis is devided into three main parts. The thesis deals with the definition of basic terms in the first theoretical part. In the second part, the thesis deals with a solution design of numerical and sorting algorithms using pseudocodes, flowcharts, tables, C language. In the last part, all algorithm solutions are summarized in a table and implemented on the MSP430 platform.

Key words: Algorithm, Flowchart, Sorting, Pseudocode

ŽILINSKÁ UNIVERZITA V ŽILINE,
FAKULTA ELEKTROTECHNIKY A INFORMAČNÝCH TECHNOLÓGIÍ
KATEDRA MECHATRONIKY A ELEKTRONIKY

ANOTAČNÝ ZÁZNAM – BAKALÁRSKA PRÁCA

Meno a priezvisko: Vincent Uhliarik **Akademický rok:** 2019/2020

Názov práce: Algoritmizácia úloh na mikroprocesorovej platforme MSP430

Počet strán:	**39**	**Počet obrázkov:**	**19**	**Počet tabuliek:**	**10**
Počet grafov:	**0**	**Počet príloh:**	**5**	**Počet použ. lit.:**	**9**

Anotácia v slovenskom jazyku:

Bakalárska práca sa zaoberá návrhom algoritmických úloh. Navrhnuté riešenia algoritmov sú implementované na mikroprocesorovú platformu MSP430. Práca sa tiež zameriava na definíciu pojmov z oblasti algoritmizácie.

Anotácia v anglickom jazyku:

Bachelor thesis deals with a algorithmic tasks. The solutions of tasks are implemented on MSP430 platform. The thesis focuses also on a definition of terms from a field of algorithms.

Kľúčové slová:

Algoritmus, Vývojový diagram, Triedenie, Pseudokód

Vedúci diplomovej práce: Doc. RNDr. Juraj Pančík, CSc.

Konzultant:

Recenzent: _____

Dátum odovzdania práce: 1.6.2020

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam elektronických príloh

Elektronická príloha č.1:..... CD\ELEKTRONICKÉ_PRÍLOHY.rar \ELEKTRONICKÉ_PRÍLOHY\Elektronická Príloha č.1

Elektronická príloha č.2:..... CD\ELEKTRONICKÉ_PRÍLOHY.rar \ELEKTRONICKÉ_PRÍLOHY\Elektronická Príloha č.2

Elektronická príloha č.3:..... CD\ELEKTRONICKÉ_PRÍLOHY.rar \ELEKTRONICKÉ_PRÍLOHY\Elektronická Príloha č.3

Elektronická príloha č.4:..... CD\ELEKTRONICKÉ_PRÍLOHY.rar \ELEKTRONICKÉ_PRÍLOHY.rar\Elektronická Príloha č.4

Elektronická príloha č.5:..... CD\ELEKTRONICKÉ_PRÍLOHY.rar \ELEKTRONICKÉ_PRÍLOHY\Elektronická Príloha č.5

POĎAKOVANIE

Ďakujem predovšetkým docentovi RNDr. Jurajovi Pančíkovi, CSc., za cenné rady, pripomienky a vedenie pri písaní tejto bakalárskej práce. Vďaka patrí aj mojej trpezlivej rodine za podporu a povzbudzovanie počas vypracovávania práce.

ČESTNÉ VYHLÁSENIE

Čestne prehlasujem, že som zadanú bakalársku prácu vypracoval samostatne, pod odborným vedením vedúceho bakalárskej práce Doc. RNDr. Juraja Pančíka, CSc. a používal som len zdroje uvedené v zozname použitej literatúry.

Súhlasím so zapožičiavaním bakalárskej práce.

Žilina 1. 6. 2020 .....

Podpis

Úvod

Predmetom bakalárskej práce je navrhnutie algoritmických úloh pre použitie na mikroprocesorovej platforme MSP430 od TI (Texas Instruments). Texas instrument je americká firma špecializujúca sa na vývoj a výrobu integrovaných obvodov a počítačovej techniky. Úlohy navrhnuté v bakalárskej práci majú poslúžiť ako výučbová pomôcka na predmete Mikroprocesorové systémy na UNIZA.

V prvej kapitole sa venujem vymedzeniu základných teoretických pojmov z oblasti algoritmizácie, ktoré sú dôležité z hľadiska riešenia ďalších častí práce.

Nasledujúca druhá časť je zameraná na konkrétne triediace a numerické algoritmy. Zaoberá sa ich popisom princípu, pseudokódom, vývojovým diagramom, kódom v jazyku C a v prípade triediacich algoritmov aj vhodnou tabuľkou znázorňujúcou triedenie konkrétneho poľa daným algoritmom. Na začiatku druhej kapitoly sú uvedené triediace algoritmy a v závere sa nachádza numerický algoritmus.

V poslednej tretej časti sú prehľadne opísané praktické riešenia algoritmických úloh aj s použitím mikroprocesorovej platformy MSP430. Výsledky práce sú zhrnuté v podobe prehľadovej tabuľky, ktorá zabezpečuje orientáciu v riešených algoritmoch popísaných v druhej kapitole.

1 TEORETICKÉ VYMEDZENIE POJMOV

1.1 Algoritmus

„Algoritmické (a analytické) myslenie je nástrojom, ktorý poskytuje svojim majiteľom silu schopnú vytvoriť z počítača poslušného otroka slepo vykonávajúceho zadané príkazy. Niežeby počítače na súčasnej úrovni boli schopné neposlušnosti, no najmä pri začínajúcich používateľoch môže nestranný pozorovateľ nadobudnúť dojem, že nie človek, ale počítač je riadiacou zložkou ich vzájomného vzťahu.“ (Skalka, a iní, 2007)

Každý študent v dnešnej dobe už určite prišiel do styku s počítačom ako používateľ a využil počítačový program. Málokto sa ale zamyslí nad tým, že tento program musel najskôr niekto vytvoriť, aby ho potom on mohol používať. Bolo potrebné presne povedať počítaču, čo a kedy má vykonať v závislosti od vstupných údajov. To znamená, že sa musel vytvoriť postup, ktorému horíme algoritmus a následne sa tento postup prepísal do programovacieho jazyka pomocou kódu, ktorému počítať rozumie. (PŠENČÍKOVÁ, 2009)

Pod pojmom algoritmus môžeme chápať návod na vykonanie činnosti, kde sa zo zadaných vstupných údajov dopracujeme v konečnom čase k výsledku. Ide o presný popis krokov, ktoré musíme uskutočniť, k dosiahnutiu výsledku. Činnosť, ktorú na základe algoritmu vykonávame, označujeme výpočet. (Skalka, a iní, 2007)

1.1.1 História slova algoritmus

Pojem algoritmus vznikol odvodením z arabského mena Muhammad ibn Musa al-Khwarizmi. Toto meno patrilo matematikovi, ktorý sa narodil na mieste, ktoré dnes poznáme ako Uzbekistan približne medzi rokmi 770 až 840. Tento muž sa zaujímal o geografiu, astronómiu a matematiku. Vďaka nemu vznikol aj ďalší matematický pojem algebra. (ABAS, 2020)

1.1.2 Vlastnosti algoritmu

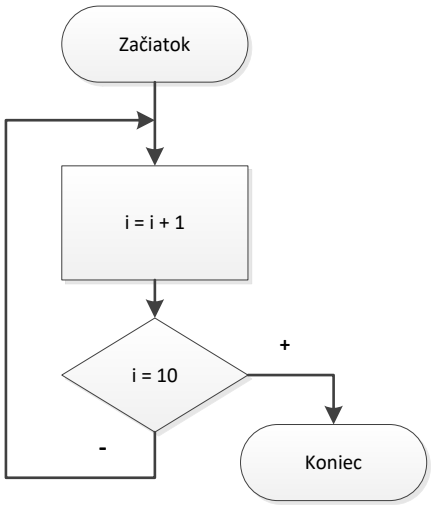
K vykonaniu určitej činnosti počítačom je potrebný presný postup, ktorý musí spĺňať (podľa (PŠENČÍKOVÁ, 2009) nasledujúce podmienky:

- musí mať začiatok a koniec
- musí byť jednoznačný
- musí byť všeobecný
- musí byť opakovateľný
- musí byť zrozumiteľný

Aj keď nám to môže byť úplne jasné, splnenie týchto podmienok je veľmi dôležité nakoľko algoritmy sú vykonávané nemysliacimi počítačmi, ktoré na rozdiel od ľudí nezbierajú skúsenosti, neučia sa z chýb, nevedia experimentovať a nedokážu si uvedomiť, ak by beh algoritmu trval až príliš dlho. (Skalka, a iní, 2007)

Aj keď sa dané podmienky zdajú byť jednoduché, tak nie vždy sa algoritmus zapíše správne. Príkladom je algoritmus zobrazený na obrázku 1.1, ktorý nemusí vždy skončiť. Daný algoritmus skončí, ak premenná i nadobudne hodnotu 10. Problém nastane vtedy, ak premenná i bude mať na začiatku hodnotu 10 alebo viac. V takomto prípade sa bude premenná inkrementovať do „nekonečna“ a program sa bude musieť ukončiť násilne alebo bude dokonca potrebné vypnúť počítač. (PŠENČÍKOVÁ, 2009)

Obrázok 1.1 Porušená podmienka konca algoritmu

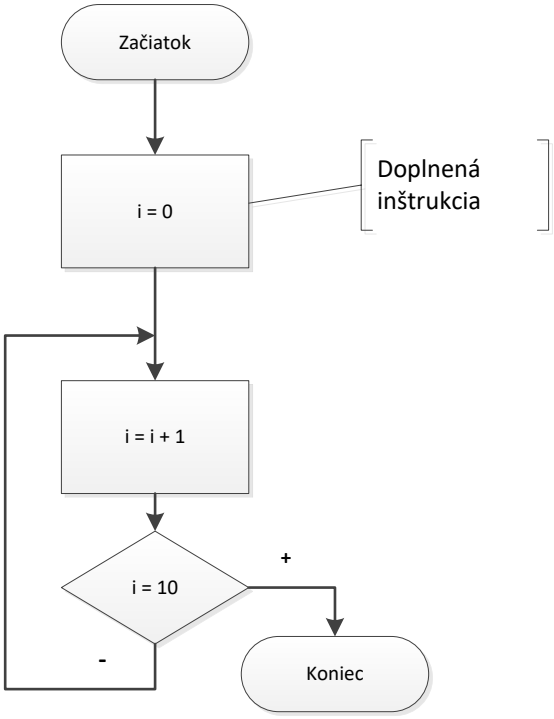


ZDROJ: vlastné spracovanie podľa: (PŠENČÍKOVÁ, 2009)

Aby sme zabezpečili koniec algoritmu, musíme na začiatku priradiť premennej i hodnotu menšiu ako 10. V takomto prípade sa bude cyklus inkrementácie opakovať

pokiaľ premenná nedosiahne hodnotu 10. Následne sa program sám ukončí. (PŠENČÍKOVÁ, 2009)

Obrázok 1.2 Splnená podmienka konca algoritmu



ZDROJ: vlastné spracovanie podľa: (PŠENČÍKOVÁ, 2009)

Môžeme si uviesť príklad z bežného života pre nekončiaci alebo dlho trvajúci algoritmus. Počítač ide uvariť jedlo podľa nasledujúceho algoritmu:

1. Hrniec s jedlom polož na varič.
2. Pusti plyn.
3. Miešaj, pokým nezačne jedlo vrieť.

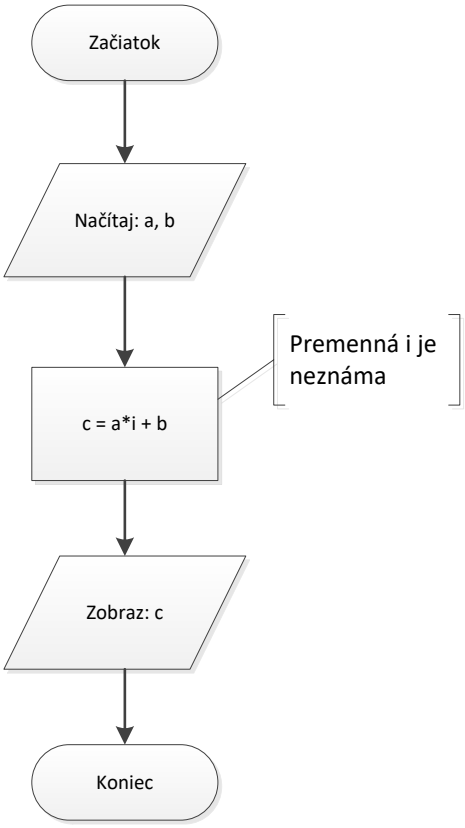
V tomto prípade môže nastať situácia, kedy vypnú plyn a jedlo nikdy nezačne vrieť. Na rozdiel od počítača by človek postupoval na základe skúsenosti a činnosť by sám ukončil, čo ale nemysliaci stroj nedokáže. (Skalka, a iní, 2007)

Pri návrhu algoritmu musíme dbať na to, že počítač je presný stroj, ktorý nevie sám rozmýšľať. Preto musí byť program a aj samotný algoritmus vytvorený tak, aby počítač v každom jednom kroku vedel, čo presne má vykonať a ktorým smerom sa má

uberať. Dobrý algoritmus by mal zahŕňať všetky možné scenáre, ktoré môžu nastať z hľadiska používateľa. (PŠENČÍKOVÁ, 2009)

Pri podmienke všeobecnosti algoritmu vyžadujeme, aby bol hromadne použiteľný pre rôzne zadané vstupné hodnoty respektíve parametre. Všeobecným algoritmom neriešime konkrétny prípad, ale skôr popisujeme postup ako sa dopracujeme k správnemu výsledku. Ako jednoduchý príklad môžeme použiť algoritmus na výpočet objemu kvádra. Pre rozmery kvádra 3, 4 a 5 vypočítame objem V = 3 . 4 . 5. Teraz ale nemáme splnenú podmienku všeobecnosti, pretože daný algoritmus platí len pre konkrétne rozmery. Všeobecne budú vstupné hodnoty zadané používateľom a uložené ako reálne čísla x, y, z. Výstupná hodnota bude reálne číslo V, ktorá bude reprezentovať objem daného kvádra. V tomto prípade by algoritmus vyzeral nasledovne: V = x . y . z, z čoho vidíme, že algoritmus sa dá použiť pre akékoľvek rozmery kvádra a výsledok dostaneme vždy správny. (Skalka, a iní, 2007)

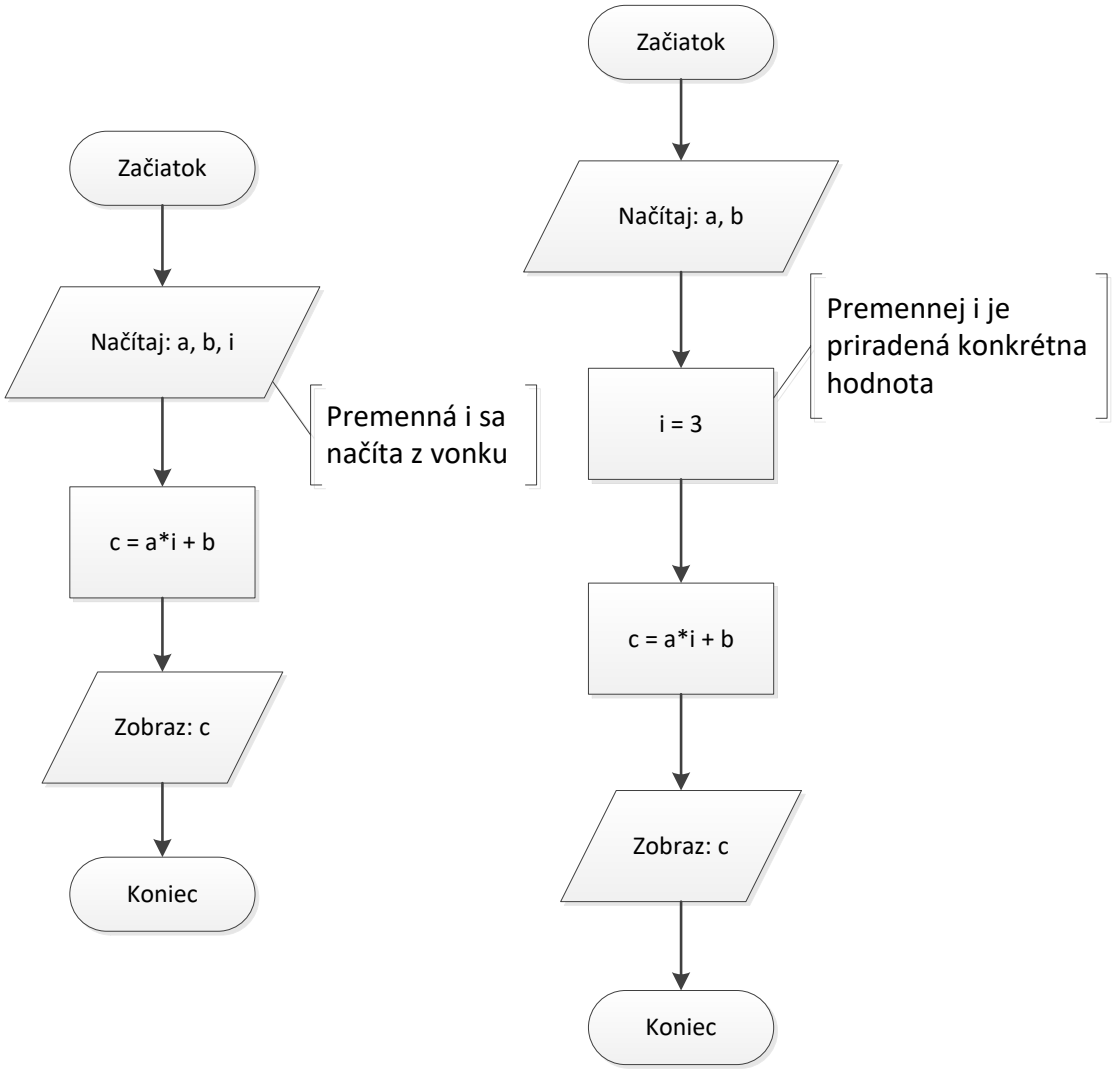
Obrázok 1.3 Porušená podmienka opakovateľnosti



ZDROJ: vlastné spracovanie podľa: (PŠENČÍKOVÁ, 2009)

Keď je algoritmus vytvorený správne, tak pri zadaní rovnakých vstupných hodnôt musíme vždy dostať ten istý výsledok. Môže však nastať situácia, ako na obrázku 1.3, kedy to tak nebude. Premenné A a B síce budú načítané zvonku, ale problémom bude premenná i, o ktorej hodnote nič nevieme. Po opakovanom spustení algoritmu pre rovnaké hodnoty A, B nemusíme vždy dostať rovnaký výsledok. (PŠENČÍKOVÁ, 2009)

Obrázok 1.4 Splnená podmienka opakovateľnosti



ZDROJ: vlastné spracovanie podľa: (PŠENČÍKOVÁ, 2009)

Ak chceme zabezpečiť opakovateľnosť tohto algoritmu, máme dve možnosti, buď premennú i načítame zvonku alebo jej priradíme konkrétnu hodnotu. Takáto situácia môže nastať v prípade, že daný algoritmus je súčasťou väčšieho celku, v ktorom

už premenná s daným názvom existuje. V tomto prípade sa zvolí iný názov premennej a potom sa upraví algoritmus podľa jednej z vyššie uvedených možností. Upravený algoritmus vidíme na obrázku 1.4. (PŠENČÍKOVÁ, 2009)

Každý algoritmus musí byť zrozumiteľný natoľko, aby ho dokázal programátor prečítať, pochopiť a potom podľa neho vytvoriť program. Musí mu taktiež rozumieť sám tvorca, pretože po istom čase môže byť požiadaný používateľom na jeho úpravu prípadne rozšírenie. Pre zápis algoritmov je preto vhodné použiť jednu z metód, ktoré sú na to určené, dostatočne používať komentáre a všetky použité premenné popísať a vysvetliť ich význam. Týmto spôsobom vytvorený algoritmus vieme kedykoľvek upraviť a rozšíriť bez toho, aby sme museli dlho rozmýšľať nad tým, ako daný algoritmus funguje a čo znamenajú jednotlivé premenné. (PŠENČÍKOVÁ, 2009)

1.1.1 Algoritmizácia

S pojmom algoritmus si často spájame aj pojem algoritmizácia. Pod algoritmizáciou chápeme aktívne vytváranie algoritmov, ktoré sú prioritne určené pre počítače inak nazývané nemysliace stroje. Algoritmizácia je dôležitou súčasťou programovania, no ani ideálne zvládnutie algoritmizácie nemusí stačiť na vytvorenie správneho algoritmu. Správny algoritmus, v prípade, ak skončí, dáva stále správne výsledky a súčasne skončí v konečnom čase pre všetky možné vstupné hodnoty. (DRLÍK, 2020)

1.2 Algoritmický jazyk

„Na to, aby sme mohli s niekým alebo niečím komunikovať, potrebujeme dorozumievací prostriedok – jazyk. Jazyk sám osebe však zvyčajne nestač. Darmo ovládate deväť cudzích jazykov, keď osoba, s ktorou sa potrebujete dohovoriť je Eskimák a jazyk, ktorý používa, sa k tým vašim svojimi výrazovými prostriedkami nepribližuje ani zďaleka.

Na komunikáciu s iným subjektom (človekom, strojom) je potrebné používať jazyk, ktorému rozumie. Pôvodne bol jazyk iba prostriedkom komunikácie medzi ľuďmi, dnes je už aj prostriedkom komunikácie medzi človekom a strojom. Jazyk,

pomocou ktorého sa komunikuje so strojom, má však oproti klasickému jazyku určité odlišnosti.“ (Skalka, a iní, 2007)

Klasický jazyk obsahuje veľa slov, z ktorých mnoho je synoným, homoným, a ďalšie slová stále pribúdajú a zanikajú. Slová môžu nadobúdať rôzne tvary a v jazyku existuje veľa výnimiek. Na komunikáciu so strojom potrebujeme jazyk stabilný, nemenný, ktorý vyžaduje presnosť, konkrétnosť a adresnosť. Tak ako existuje veľa jazykov vo svete, tak isto existuje viacero algoritmických jazykov. Tieto môžeme rozdeliť na graficky alebo textovo orientované. Medzi graficky orientované radíme vývojové diagramy a štruktúrogramy. Medzi textové patria rozhodovacie tabuľky, slovný zápis algoritmu a programovacie jazyky. (Skalka, a iní, 2007)

1.2.1 Slovné vyjadrenie

Tento spôsob zápisu algoritmu používame bežne v živote a pritom si to ani neuvedomujeme. Jedná sa o rozličné návody na obsluhu, recepty v kuchárskej knihe, technologické postupy a podobne. Hlavnou výhodou zápisu algoritmu slovným vyjadrením je tá, že sa s ňou dohovoríme aj s laikom, ktorý nemá programátorské zručnosti. Používa sa tiež pri komunikácií medzi programátorom a užívateľom, kedy programátor potrebuje zistiť všetky detaily, ktoré má objednaný program obsahovať. Následne si slovný zápis algoritmu nechá užívateľom schváliť alebo doplniť . Môžeme povedať, že to je asi jediný spôsob použiteľný na komunikáciu medzi touto dvojicou ľudí. Na druhú stranu je slovný zápis najmenej prehľadný zo všetkých foriem zápisov, ťažko sa v ňom orientuje a tiež sa nedá z určitosťou povedať, že spĺňa podmienky dobrého algoritmu. (PŠENČÍKOVÁ, 2009)

1.2.2 Vývojové diagramy

Už pri vyššie popisovaných podmienkach algoritmu sme sa stretli s vývojovými diagramami. Zápis algoritmov pomocou vývojových diagramov je široko rozšírený medzi analytikmi a programátormi. Vďaka jeho jasnej prehľadnosti sa v ňom nemá problém zorientovať človek so základnými programátorskými znalosťami. Radíme ho medzi jednu z najdokonalejších foriem zápisu algoritmov a môžeme ho nazvať symbolický algoritmický jazyk. (PŠENČÍKOVÁ, 2009)

Vývojové diagramy pozostávajú z konkrétnych, jednoznačných symbolov, ktoré sú medzi sebou prepojené čiarami so šípkami. Šípky určujú smer, ktorým sa uberá algoritmus. Medzi symboly vývojových diagramov patria:

- ovály - označujú začiatok alebo koniec programu
- obdĺžniky - znázorňujú istú činnosť programu (matematické alebo logické operácie)
- kosoštvorce - obsahujú v sebe podmienku podľa ktorej sa program rozvetví, ak je splnená podmienka, program pokračuje vetvou so znamienkom +, ak nie, tak pokračuje opačnou vetvou so znamienkom -
- kosodĺžniky – využívajú sa na komunikáciu s počítačom, buď slúžia na načítanie dát potrebné pre beh program alebo na zobrazenie dát na výstupe (PŠENČÍKOVÁ, 2009)

Existuje viac symbolov, ktoré sa používajú pri zápise algoritmov pomocou vývojových diagramov. Keďže v tejto práci nebudem používať zložité vývojové diagramy, tak ich popis nepovažujem za nutný.

1.2.3 Štruktúrogramy

Zápis pomocou štruktúrogramov sa vo veľkej miere podobá na zápis pomocou vývojových diagramov. Štruktúrogramy sú zložené zo základných algoritmických štruktúr, ktoré sa vnárajú do seba alebo radia za sebou do postupnosti. Od vývojových diagramov sa odlišujú tým, že nie sú definované normou ako by sa mali správne používať. (Skalka, a iní, 2007)

1.2.4 Rozhodovacie tabuľky

Ďalším z možných zápisov algoritmov sú rozhodovacie tabuľky. Typický príklad rozhodovacej tabuľky, z ktorým sa stretol každý, je rozvrh hodín. Zápis algoritmu pomocou tabuliek využívame vtedy, keď máme viac možností s vlastným riešením, ktoré sa dajú jednoducho popísať. V tabuľke 1.1 vidíme riešenie logického súčinu a súčtu pre dve logické premenné A a B. Keď vezmeme do úvahy rozvrh hodín a tabuľku logických operácií, môžeme povedať, že zápis algoritmov pomocou tabuliek

je veľmi prehľadný a vhodný pre pomerne veľké množstvo prípadov. Pri použití tabuliek si ale treba dať pozor na to, aby popis v jednotlivých bunkách nebol zložitý a nebolo potrebné ďalšie vysvetlenie. Potom by tabuľka stratila svoju prehľadnosť a priblížila sa skôr k slovnému zápisu. (PŠENČÍKOVÁ, 2009)

Tabuľka 1.1 Riešenie logických operácií AND a OR

A	B	A and B	A or B
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

ZDROJ: vlastné spracovanie podľa: (PŠENČÍKOVÁ, 2009)

„Rozhodovacie tabuľky nepopisujú činnosti v poradí, v akom sa majú vykonávať, ale definujú v tabuľke to, čo sa má robiť pre rôzne kombinácie hodnôt premenných.“ (Skalka, a iní, 2007)

1.2.5 Zápis v jazyku C

Akokoľvek by sme sa snažili vložiť do počítača algoritmus v tvare vývojového diagramu alebo iného tvaru, tak by si s tým v dnešnej dobe neporadil. Možno raz v budúcnosti áno, ale tam sme ešte nedospeli. Aby počítač daný algoritmus „pochopil“, musí ho programátor prepísať do niektorého z programovacích jazykov, vyladiť ho a nakoniec preložiť do strojového kódu. Algoritmus prepísaný do programovacieho jazyka sa volá program. Na to, aby si počítač poradil s programom, musí byť počítač vybavený prekladačom. Ten slúži na preloženie programu napísaného v programovacom jazyku do strojového kódu. V dnešnej dobe sa už s tým príliš nemusíme zaoberať nakoľko prekladač býva súčasťou integrovaného vývojového programátorského prostredia. (PŠENČÍKOVÁ, 2009)

Ďalej v práci budem používať programovací jazyk C, preto je vhodné si o ňom niečo najskôr povedať.

Jazyk C sa považuje za univerzálny jazyk, ktorý je jednoduchý a flexibilný. Aj preto sa používa vo veľkom množstve aplikácií. Koncept jazyka C vieme nájsť v ďalších programovacích jazykoch a preto ak sa naučíme programovať v jazyku C, tak

ľahko pochopíme aj podobné jazyky založené na koncepte jazyka C. Významnou črtou jazyka C je, že sa môže sám rozšíriť pomocou knižníc. Knižnice obsahujú v sebe rôzne funkcie, ktoré môžeme používať. Pre využitie týchto funkcií musíme zahrnúť na začiatok nášho programu dotyčnú knižnicu. Programovať v jazyku C sa môže naučiť každý, pretože vývojové prostredia sú voľne dostupné a existuje ich viacero. Jazyk C patrí medzi najpoužívanejšie programovacie jazyky na svete a preto ak si neviete z niečím sami pomôcť nie je problém nájsť na internete fórum, kde je dotyčný problém a riešenie popísané, alebo ak nie je, tak si sami môžete popýtať radu. (Guru99, 2020)

„Každý jazyk je založený na nejakom slovníku. Jeho prvky sa obyčajne nazývajú slová; v teórií formálnych jazykov im hovoríme (základné) symboly. Pre každý jazyk je charakteristické, že určité postupnosti slov sa považujú za správne, dobre vytvorené vety tohto jazyka, pričom iné sa považujú za nesprávne alebo zle vytvorené. Na základe čoho sa dá rozhodnúť, či daná postupnosť slov je správna veta alebo nie? Je to gramatika, syntax alebo štruktúra jazyka. Všeobecne definujeme syntax ako množinu pravidiel alebo formúl, ktorá definuje množinu (formálne konkrétnych) viet.“ (Wirth, 1975)

Zo slov Niklausa Wirtha je zrejmé, že aj jazyk C má svoju syntax, svoju štruktúru, ktorú musíme dodržať pre správny chod programu. Nemôžeme používať pre názvy premenných rezervované slová, ktoré sú jasne zadefinované. Jazyk C má 32 takýchto rezervovaných slov. V anglickom jazyku ich vieme vyhľadať pod názvom „reserved words“ alebo tiež „keywords“.

1.3 Výpočtová zložitosť

Zložitosť algoritmu je meraná množstvom spotrebovanej výpočtovej práce počítačom, keď sa rieši istý problém pomocou daného algoritmu. Výpočtová zložitosť sa vzťahuje na prevádzkový čas, počet výpočtových krokov alebo na pamäťové miesto. My budeme považovať výpočtovú zložitosť algoritmu ako funkciu veľkosti vstupných dát. Dĺžka úlohy je n, ak algoritmus pracuje s n-prvkovou množinou. Na vyriešenie problému väčšinou existuje viac ako jeden algoritmus. Tie sa líšia len niektorými parametrami.

Vo všeobecnosti existuje viacero typov zložitosti. Najčastejšie sa zložitosť algoritmov uvádza ako worst-case a average-case zložitosť. Pri zložitosti worst-case sa zameriavame na meranie zložitosti algoritmu pre najhorší možný prípad, ktorý pre dané parametre môže nastať. Zložitosť average-case udáva priemerný počet potrebných krokov na vyriešenie problému daným algoritmom.

Ak je zložitosť algoritmu polynómom vstupnej veľkosti n alebo je ohraničená polynómom, tak hovoríme o efektívnom alebo tiež polynomiálnom algoritme. Zložitosti efektívnych algoritmov sú napríklad $4n - 3$, $n^3 + 2$. Ak sú zložitosti 3^n alebo $n!$, tak považujeme algoritmy za neefektívne.

Keď existuje algoritmus, ktorým vyriešime daný problém, tak problém nazývame zvládnuteľný. Ak sa však môže dokázať, že na riešenie problému neexituje efektívny algoritmus, tak sa problém nazýva nezvládnuteľný alebo tiež beznádejný. (ABAS, 2020)

Pre určenie výpočtovej zložitosti daného algoritmu môžeme použiť nasledujúci postup:

1. Určíme spôsob, akým sa meria veľkosť vstupu. Či veľkosť vstupu závisí od počtu vstupných údajov alebo od ich hodnôt.
2. Určíme počet f(n) vykonaných elementárnych operácií na vstupe s veľkosťou n. Ak nedokážeme presne určiť tento počet, tak sa snažíme nájsť čo najlepší horný odhad.
3. Výsledná formula f(n) je súčtom viacerých členov. Pre nás je dôležitý ten, ktorý rastie najrýchlejšie a teda ostatné členy zanedbáme.
4. Vyškrtáme konštanty, ktorými sa násobí formula.

Dajme si príklad kedy sa v behu algoritmu vykoná $3n^2 + 4n$ elementárnych operácií. Najskôr vyškrtneme pomalšie rastúci člen $4n$. Ostane nám $3n^2$, kde následne vyškrtneme konštantu 3 a získame funkciu n^2. Výslednú funkciu $g(n) = n^2$, nazveme asymptotickou výpočtovou zložitosťou a môžeme povedať, že algoritmus má časovú zložitosť $O(g(n))$. Pre náš príklad to teda bude $O(n^2)$.

Algoritmy môžu mať zložitosť zapísanú komplikovanou funkciou, no najčastejšie sa stretneme so zložitosťami lineárnymi O(n), kvadratickými $O(n^2)$, logaritmickými O(log n), exponenciálnymi $O(2^n)$ a konštantnými O(1). Výpočtová zložitosť má veľmi dôležitú úlohu. V tabuľke 1.2 vidíme závislosť počtu elementárnych operácií od veľkosti n pre konkrétne zložitosti. Medzi efektívne algoritmy radíme tie so zložitosťou O(log n), O(n), O(n log n) a všeobecne $O(n^k)$, kde k je pevná konštanta s čo najmenšou hodnotou.

Tabuľka 1.2 Chovanie rôznych zložitostí algoritmov

Funkcia	n = 10	n = 100	n = 1000	n = 10 000
log n	1	2	3	4
n	10	100	1000	10 000
n log n	10	200	3000	40 000
$\mathbf{n^2}$	100	10 000	10^6	10^8
$\mathbf{n^3}$	1000	10^6	10^9	10^{12}
$\mathbf{2^n}$	1024	$\approx 10^{31}$	$\approx 10^{310}$	$\approx 10^{3100}$

ZDROJ: vlastné spracovanie podľa: (Mareš, a iní, 2017)

1.4 Definícia triediacich algoritmov

Dobre vieme, že v dátach, ktoré sú usporiadané istým spôsobom sa ľahšie orientuje a hľadá ako v neusporiadaných. Z tohto dôvodu sa nezaobíde bez triedenia a triediacich programov žiadna úloha hromadného spracovania dát. (PŠENČÍKOVÁ, 2009)

„Povestnou čerešničkou na torte algoritmizácie sú triediace algoritmy.“ (PŠENČÍKOVÁ, 2009)

„Pod triedením zvyčajne rozumieme proces preusporiadania danej množiny objektov v špecifickom poradí. Účelom triedenia je uľahčiť neskoršie vyhľadávanie prvkov triedenej množiny.“ (Wirth, 1975)

Napriek tomu, že klasické slovníky definujú triedenie ako proces delenia podľa rôznych tried, druhu alebo inej charakteristickej črty, programátori používajú tento pojem v zmysle usporiadania prvkov danej množiny vzostupne alebo zostupne. (KNUTH, 1998)

Už od mala nás učili dávať si veci do poriadku a teda s istým triedením sme sa už všetci streli. Objekty sa triedia všade tam, kde sa budú neskôr vyhľadávať a vyberať, napríklad v telefónnych zoznamoch, obsahoch kníh, knižniciach, slovníkoch, skladoch. Triedenie teda považujeme za základnú a dôležitú činnosť pri spracovaní údajov. Využíva sa hlavne na to, aby sa poukázalo na veľkú rozmanitosť algoritmov. Všetky tieto algoritmy majú rovnaký cieľ, no každý jeden je optimálny v istom zmysle, pre určitý typ triedenia. (Wirth, 1975)

1.5 Spôsoby delenia triediacich algoritmov

„Z pohľadu vlastného spôsobu zoraďovania dát existuje veľa rôznych metód, ktoré sa líšia:

- zložitosťou algoritmu
- nárokom na pamäť (počtom pomocných premenných)
- rýchlosťou (počtom porovnaní)“ (PŠENČÍKOVÁ, 2009)

Obvykle sa metódy triedenia delia do dvoch kategórií:

- triedenie polí - vnútorné triedenie
- triedenie súborov – vonkajšie triedenie

Vnútorným triedením chápeme triedenie polí, ktoré sú uložené v rýchlych vnútorných pamätiach počítačov a prístup k nim je náhodný. Vonkajším triedením máme na mysli triedenie súborov, ktoré sú síce uložené na priestrannejších vonkajších pamätiach no na druhej strane sú tieto pamäte pomalšie. (Wirth, 1975)

Pri metódach triedenia polí by sme si mali predovšetkým dávať pozor na úsporné využívanie pamäti, ktorú máme. Znamená to, že permutácia usporadúvania prvkov sa musí vykonať na mieste a metódy, pomocou ktorých sa prvky z počiatočného poľa premiestňujú do výsledného poľa, nie sú z pohľadu pamäte tak zaujímavé. Vykonávať niečo na mieste znamená v našom použití vykonávať triedenie bez pomocnej pamäti. Kritériom úspory pamäti sme zúžili výber metód spomedzi možných riešení. Ďalej prejdeme ku klasifikácií metód podľa efektívnosti, teda podľa časovej náročnosti. Efektívnosť danej metódy vieme dobre určiť spočítaním počtov porovnaní

a presunov prvkov v poli potrebných na usporiadanie. Spomenuté počty sú závislé od počtu prvkov obsiahnuté v poli, ktoré treba roztriediť. Algoritmy, ktoré vyžadujú rádovo *n . log n* porovnaní, považujeme za dobré algoritmy. Rozoberieme si niekoľko jednoduchých a jasných metód triedení, ktoré voláme priame metódy. Priame metódy potrebujú rádovo n^2 porovnaní. Dôvody prečo sa najskôr zaoberať priamymi metódami sú tri:

- na objasnenie hlavných charakteristických čŕt triedenia sú osobitne vhodné
- programy pre priame metódy sú krátke a ľahko pochopiteľné – netreba zabudnúť, že v pamäti zaberajú miesto aj programy
- dômyselné metódy majú síce menej operácií, ale sú zložitejšie v podrobnostiach, preto pre malé n sú rýchlejšie priame metódy, no pri veľkých n sa už nepoužívajú

Priame metódy možno rozdeliť na tri základné skupiny:

- triedenie vkladaním
- triedenie výberom
- triedenie výmenou (Wirth, 1975)

Priame metódy majú svoje výhody, sú jednoduché, krátke a triedia na mieste. Na druhú stranu je tu veľká nevýhoda a tou je výpočtová zložitosť $O(n^2)$. Z toho vyplýva, že priame metódy triedenia sú použiteľné v prípadoch, kedy sa nejdená o veľké množstvo triedených dát. (Mareš, a iní, 2017)

2 PREHĽAD VYBRANÝCH ALGORITMOV

V nasledujúcej kapitole sa budem venovať popisu konkrétnych algoritmov. Pri každom triediacom algoritme budem uvádzať príklad triedenia poľa celých čísel 6, 2, 4, 8, 8, 3, 1, 3, 9, 5 vzostupne (od najmenšieho po najväčšie). Pri jednotlivých algoritmoch spomeniem opis, pseudokód, vývojový diagram, tabuľky s riešením triedenia vyššie uvedeného poľa a tiež riešenie algoritmu pomocou kódu v jazyku C.

Pri numerickom algoritme popíšem výpočet goniometrickej funkcie sínus slovne, cez pseudokód a následne v jazyku C vytvoreným kódom.

2.1 Bubblesort

Bublinkové triedenie (Bubblesort) nám už z názvu hovorí v čom spočíva jeho základná myšlienka. Ide o to, aby sme nechali prebublať väčšie prvky v poli rovnako ako stúpajú bublinky v nápoji. V triediacom algoritme sa opakovane prechádza celé pole prvkov, kde sa vzájomne porovnávajú susedné dvojice. Ak daná dvojica nie je správne zoradená, tak sa prvky vymenia. V prípade, že prvky sú usporiadané správne, tak ostanú na svojich pozíciách. (Mareš, a iní, 2017)

Tento základný postup bublinkového triedenia sa dá vylepšiť jednoducho tým, že si zapamätáme, či nastala alebo nenastala výmena prvkov v poli. Týmto dosiahneme, že porovnávanie a teda aj celý algoritmus skončí v momente, kedy pri prechode poľa nenastala výmena prvkov. (Wirth, 1975)

Algoritmus bublinkového triedenia môžeme zapísať jednoduchým pseudokódom, ktorý vidíme na obrázku 2.1.

Obrázok 2.1 Pseudokód - Bubblesort

Vstup: Pole P[n]

1. pokračuj ← 1
2. Pokiaľ je pokračuj = 1:
3. pokračuj ← 0
4. Pre i = 0,...,n-2:
5. Ak je P[i] > P[i+1]:
6. Prehoď prvky P[i] a P[i+1]
7. pokračuj ← 1

Výstup: Usporiadané pole P

ZDROJ: vlastné spracovanie podľa (Mareš, a iní, 2017)

Keď sa pozrieme na pseudokód, tak vidíme, že prechod krokmi 4 až 7 ide cez všetky prvky pola. Z toho vyplýva, že výpočtová zložitosť bublinkového triedenia bude minimálne O(n) a to pre najlepší možný prípad. Pre najhorší možný prípad budeme musieť prejsť cez pole n krát a teda zložitosť pre najhorší prípad bude $O(n^2)$. (Mareš, a iní, 2017)

Tabuľka 2.1 Bublinkové triedenie - prvý prechod

6	2	4	8	8	3	1	3	9	5
2	**6**	4	8	8	3	1	3	9	5
2	4	6	**8**	8	3	1	3	9	5
2	4	6	8	**8**	3	1	3	9	5
2	4	6	8	**8**	3	1	3	9	5
2	4	6	8	3	**8**	1	3	9	5
2	4	6	8	3	1	**8**	3	9	5
2	4	6	8	3	1	3	8	**9**	5
2	4	6	8	3	1	3	8	**9**	5
2	4	6	8	3	1	3	8	5	9

ZDROJ: vlastné spracovanie

Prvý prechod poľom desiatich čísel bublinkovým triedením môžeme vidieť v tabuľke 2.1. Počiatočný tvar poľa sa nachádza v prvom riadku. Červené čísla tvoria dvojicu čísel, ktoré sa aktuálne porovnávajú a väčšie z nich je tučné. Ak sú porovnávané

čísla v nesprávnom poradí, tak sa v nasledujúcom riadku vymenia a následne sa porovnáva ďalšia dvojica čísel. Tento postup sa opakuje a po skončení prvého prechodu poľom (posledný riadok) sa na posledné miesto dostane číslo s najväčšou hodnotou, v mojom prípade číslo 9.

V nasledujúcej tabuľke 2.2 vidíme usporiadanie čísel po i-tom prechode. Číslo, ktoré sa dostalo na správnu pozíciu v danom cykle je červené a tučné. Môžeme si všimnúť, že po 6. prechode už bolo pole usporiadané, no aj napriek tomu sa znovu porovnalo celé pole. Keď sa pozrieme na pseudokód bublinkového triedenia, tak vidíme, že premennej pokračuj sa priradí hodnota 1 vždy, keď nastane v danom prechode aspoň jedna výmena a porovnávanie sa uskutočňuje pokým je premenná pokračuj rovná 1. Takže vždy po prechode, kedy sú už všetky čísla na správnych pozíciách sa vykoná ešte jedno porovnávanie. Môžeme ho nazvať aj kontrolným prechodom.

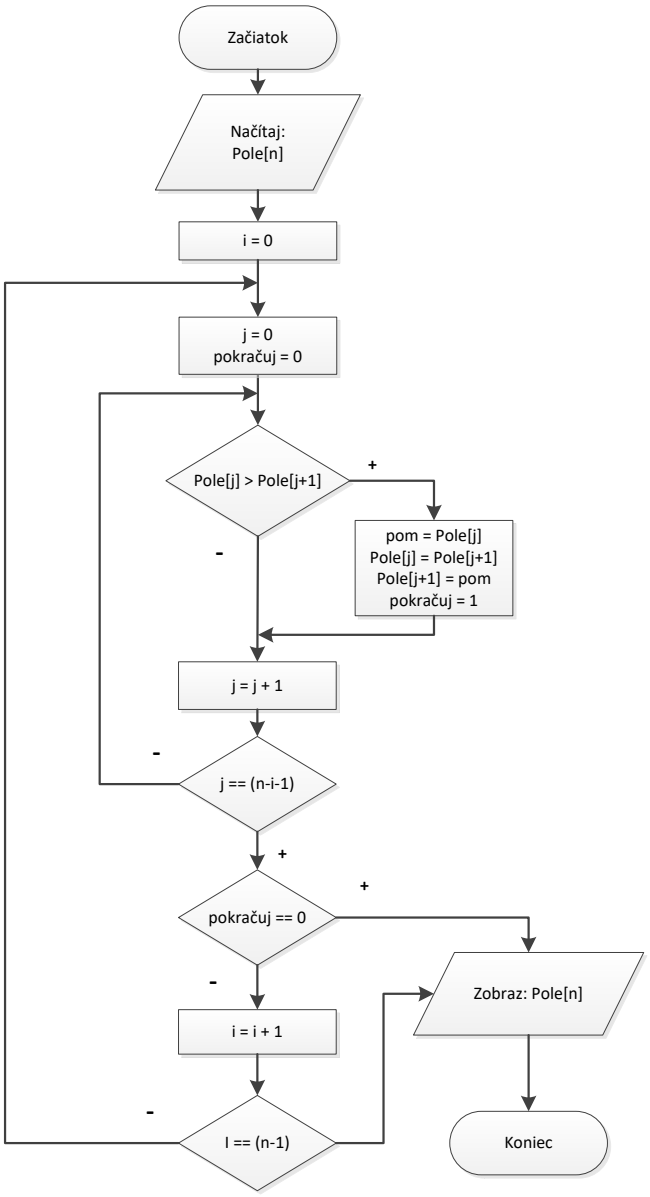
Tabuľka 2.2 Bublinkové triedenie - i-tý prechod

Prechod	Usporiadanie poľa po i-tom prechode									
1	2	4	6	8	3	1	3	8	5	**9**
2	2	4	6	3	1	3	8	5	**8**	9
3	2	4	3	1	3	6	5	**8**	8	9
4	2	3	1	3	4	5	**6**	8	8	9
5	2	1	3	3	4	**5**	6	8	8	9
6	1	2	3	3	**4**	5	6	8	8	9
7	1	2	3	3	4	5	6	8	8	9

ZDROJ: vlastné spracovanie

Kvôli lepšej prehľadnosti je lepšie prepísať pseudokód do vývojového diagramu a až potom vytvoriť program v jazyku C. Prepísaný pseudokód bublinkového triedenia do vývojového diagramu vidíme na obrázku 2.2.

Obrázok 2.2 Vývojový diagram - Bublesort



ZDROJ: Vlastné spracovanie podľa (PŠENČÍKOVÁ, 2009)

Podľa vyššie uvedeného vývojového diagramu sa vytvorí zdrojový kód v jazyku C (obrázok 2.3), s ktorým si už počítač poradí za pomoci vhodného prekladača.

Obrázok 2.3 Algoritmus Bubblesort v jazyku C

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.  int Pole[100],n,i,j,pom,pokracuj;          //deklarácia poľa a premenných
5.  printf("Zadaj velkost pola:\n");
6.  scanf("%d", &n);                           //načítanie veľkosti poľa do premennej n
7.  printf("Zadaj prvky pola[%d]:\n", n);
8.  for(i = 0; i < n; i++)
9.          scanf("%d", &Pole[i]);             //načítanie prvkov poľa
10. for(i = 0; i < n-1; i++)
11. {
12.         pokracuj = 0;
13.         for(j = 0; j < n-i-1; j++)
14.         {
15.                 if (Pole[j] > Pole[j+1])   //podmienka, či sa majú prvky vymeniť
16.                 {
17.                         pom = Pole[j];
18.                         Pole[j] = Pole[j+1];
19.                         Pole[j+1] = pom;
20.                         pokracuj = 1;
21.                 }
22.         }
23.         if(pokracuj == 0)
24.                 break;
25. }
26. printf("\nUsporiadane pole:\n");
27. for(i = 0; i < n; i++)
28.         printf("%d  ",Pole[i]);            //výpis usporiadaného poľa
29. return 0;
30. }
```

ZDROJ: vlastné spracovanie

2.1.1 Shakesort

Na bublinkové triedenie priamo nadväzuje triedenie pretriasaním (Shakesort), ktorého názov pochádza z prirovnania k barmanovi ako trasie šejkrom. Zatrasením šejkrom sa bublinky v nápoji dostávajú raz na jednu a raz na druhú stranu. Rovnakým spôsobom sa prechádza poľom prvkov. Najskôr zľava doprava dostaneme najväčší prvok a následne dostaneme najmenší prvok sprava doľava. Pole prvkov sa teda bude

usporadúvať aj od konca aj od začiatku. Posledná utriedená časť bude stred poľa. V každom cykle sa taktiež zaznamenáva pozícia prvku, ktorý sa dostal na správne miesto a v nasledujúcom cykle sa už daný prvok nebude porovnávať. (PŠENČÍKOVÁ, 2009)

Pre triedenie pretriasaním musíme doplniť algoritmus bublinkového triedenia o prechod poľa v opačnom smere. Takto doplnený pseudokód vidíme na obrázku 2.4.

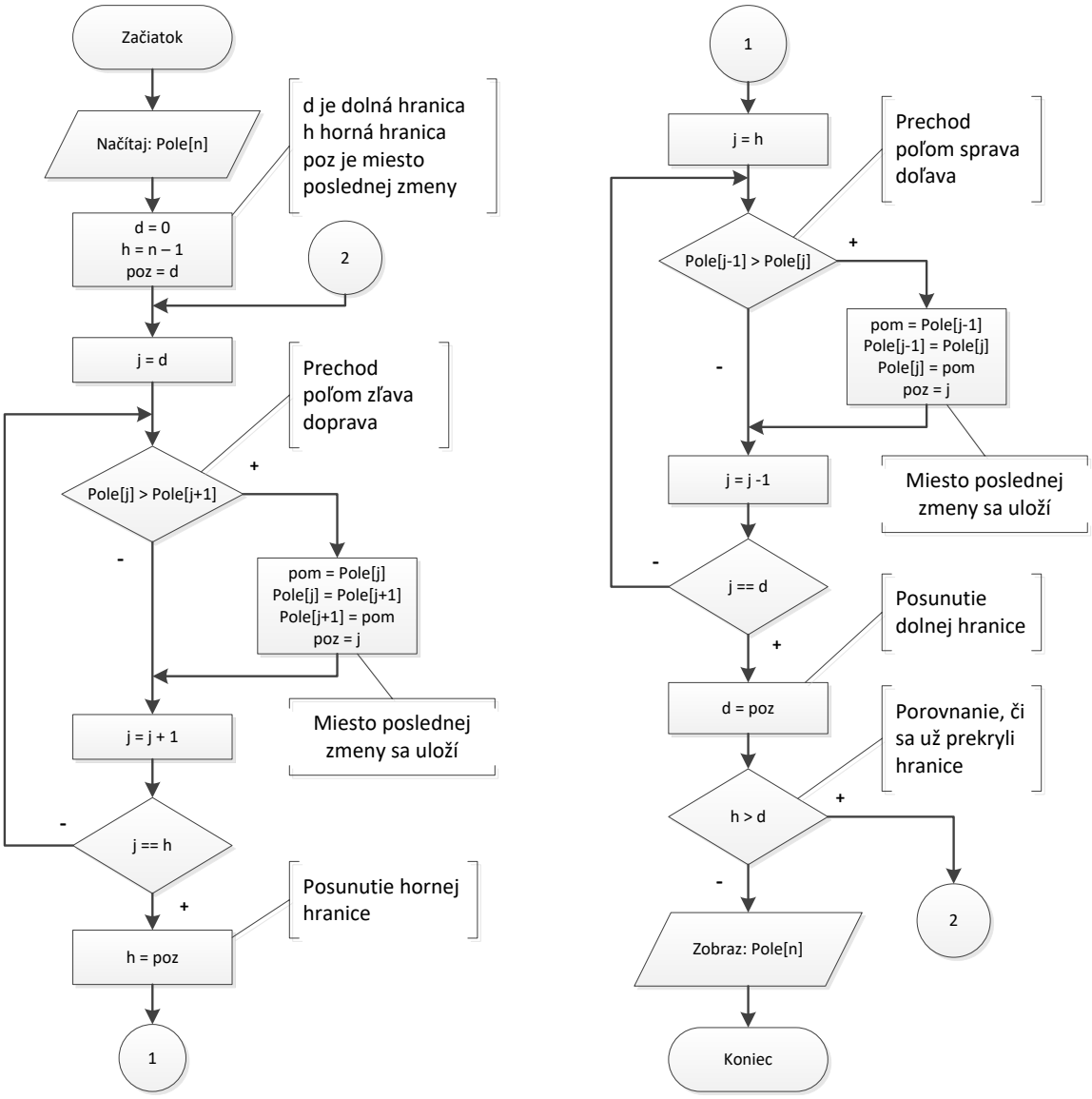
Obrázok 2.4 Pseudokód Shakesort

```
Vstup: Pole P[n]
1.  d ← 0
2.  h ← n – 1
3.  pozícia ← d
4.  Pokiaľ je h > d:
5.      Pre j = d,...,h-1:
6.          Ak je P[j] > P[j+1]:
7.              Prehoď prvky P[j] a P[j+1]
8.              pozícia ← j
9.      h ← pozícia
10.     Pre j = h,...,d+1:
11.         Ak je P[j-1] > P[j]:
12.             Prehoď prvky P[j-1] a P[j]
13.             pozícia ← j
14.     d ← pozícia
Výstup: Usporiadané pole P
```

ZDROJ: vlastné spracovanie

Vývojový diagram pre triedenie pretriasaním vytvoríme rovnako ako pri bublinkovom triedení z vyššie uvedeného pseudokódu. Vytvorený vývojový diagram s príslušným popisom sa nachádza na obrázku 2.5. Diagram síce vyzerá o dosť zložitejší ako pri bublinkovom triedení, ale v princípe je iba predĺžený o prechod poľom sprava doľava.

Obrázok 2.5 Vývojový diagram – Shakesort



ZDROJ: Vlastné spracovanie podľa (PŠENČÍKOVÁ, 2009)

V tabuľke 2.3 a 2.4 vidíme prvý a druhý prechod poľom. Prvý prechod prechádza poľom zľava doprava a je totožný s prvým prechodom v bublinkovom triedení, kedy sa na poslednú pozíciu dostane najväčšia hodnota, v našom prípade 9. Druhý prechod už ide opačným smerom a na prvú pozíciu sa dostane číslo 1. Čísla sa vymieňajú rovnakým spôsobom ako v prípade bublinkového triedenia s využitím jednej pomocnej premennej.

Tabuľka 2.3 Triedenie pretriasaním - prvý prechod

6	2	4	8	8	3	1	3	9	5
2	**6**	4	8	8	3	1	3	9	5
2	4	6	**8**	8	3	1	3	9	5
2	4	6	8	**8**	3	1	3	9	5
2	4	6	8	**8**	3	1	3	9	5
2	4	6	8	3	**8**	1	3	9	5
2	4	6	8	3	1	**8**	3	9	5
2	4	6	8	3	1	3	8	**9**	5
2	4	6	8	3	1	3	8	**9**	5
2	4	6	8	3	1	3	8	5	9

ZDROJ: vlastné spracovanie

Tabuľka 2.4 Triedenie pretriasaním - druhý prechod

2	4	6	8	3	1	3	**8**	5	**9**
2	4	6	8	3	1	3	**5**	8	**9**
2	4	6	8	3	1	**3**	5	8	**9**
2	4	6	8	**3**	1	3	5	8	**9**
2	4	6	**8**	1	3	3	5	8	**9**
2	4	**6**	1	8	3	3	5	8	**9**
2	**4**	1	6	8	3	3	5	8	**9**
2	1	4	6	8	3	3	5	8	**9**
1	2	4	6	8	3	3	5	8	**9**

ZDROJ: vlastné spracovanie

V nasledujúcej tabuľke 2.5 vidíme tvar poľa po každom prechode algoritmu. Prechody sa striedajú jeden krát doprava, jeden krát doľava dovtedy kým nie sú všetky prvky s istotou na správnych pozíciách. Čísla, ktoré sa dostali po danom prechode na správnu pozíciu sú vyznačené červenou. Modrou sú vyznačené čísla, ktoré sa už ďalej v nasledujúcich prechodoch poľa neporovnávajú. V niektorých riadkoch sa dostali na správne miesto až dve čísla. To je spôsobené tým, že v algoritme Shakesort sa zaznamenáva pozícia poslednej výmeny. Keďže za touto pozíciou už neprebehla výmena prvkov v poli, môžeme povedať, že za ňou je už pole usporiadané.

Tabuľka 2.5 Triedenie pretriasaním - i-tý prechod

Prechod	Usporiadanie poľa po i-tom prechode									
1	2	4	6	8	3	1	3	8	5	**9**
2	**1**	2	4	6	8	3	3	5	8	**9**
3	**1**	2	4	6	3	3	5	**8**	**8**	**9**
4	**1**	**2**	**3**	4	6	3	5	**8**	**8**	**9**
5	**1**	**2**	**3**	4	3	5	**6**	**8**	**8**	**9**
6	**1**	**2**	**3**	**3**	4	5	**6**	**8**	**8**	**9**
7	**1**	**2**	**3**	**3**	**4**	**5**	**6**	**8**	**8**	**9**

ZDROJ: vlastné spracovanie

Kód v jazyku C pre algoritmus Shakesort sa vytvorí spôsobom ako pri Bubblesorte za pomoci vývojového diagramu. Kód je rozšírený o prechod poľa sprava doľava a tiež o zaznamenávanie pozície poslednej výmeny. V kóde pre Shakesort sa už ale nenachádza premenná pokračuj nakoľko tu už nie je potrebná. Zdrojový kód v jazyku C vidíme na obrázku 2.6.

Vylepšenie bublinkového triedenia o prechádzanie poľa obomi smermi však nijako neovplyvňuje počet výmen. Vylepšením sa len znížil počet nadbytočných dvojnásobných kontrol. Výmena dvoch čísel navzájom je väčšinou oveľa nákladnejšou operáciou ako samotné porovnanie a preto vylepšenie nemá až taký účinok ako by sa nám mohlo zdať. Algoritmus triedenia pretriasaním je výhodné použiť hlavne v prípadoch, kedy vieme, že prvky už sú takmer usporiadané. No v praxi to je len zriedkavý prípad. V podstate všetky priame metódy posúvajú prvok iba o jednu pozíciu v každom základnom kroku a preto ich výpočtová zložitosť často dosahuje hodnotu $O(n^2)$. (Wirth, 1975)

Obrázok 2.6 Algoritmus Shakesort v jazyku C

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.  int Pole[100],n,d,h,i,j,pom,poz;          //deklarácia poľa a premenných
5.  printf("Zadaj velkost pola:\n");
6.  scanf("%d", &n);                          //načítanie veľkosti poľa do premennej n
7.  printf("Zadaj prvky pola[%d]:\n", n);
```

```
8.  for(i = 0; i < n; i++)
9.          scanf("%d", &Pole[i]);                  //načítanie prvkov poľa
10. d = 0;                                          //stanovenie dolnej hranice
11. h = n - 1;                                      //stanovenie hornej hranice
12. poz = d;
13. while(h > d)
14. {
15.         for(j = d; j < h; j++)
16.         {
17.                 if (Pole[j] > Pole[j+1])        //podmienka, či sa majú prvky vymeniť
18.                 {
19.                         pom = Pole[j];
20.                         Pole[j] = Pole[j+1];
21.                         Pole[j+1] = pom;
22.                         poz=j;
23.                 }
24.         }
25.         h = poz;                                //posunutie hornej hranice
26.         for(j = h; j > d; j--)
27.         {
28.                 if (Pole[j-1] > Pole[j])        //podmienka, či sa majú prvky vymeniť
29.                 {
30.                         pom = Pole[j-1];
31.                         Pole[j-1] = Pole[j];
32.                         Pole[j] = pom;
33.                         poz=j;
34.                 }
35.         }
36.         d = poz;                                //posunutie dolnej hranice
37. }
38. printf("\nUsporiadane pole:\n");
39. for(i = 0; i < n; i++)
40.         printf("%d  ",Pole[i]);                 //výpis usporiadaného poľa vzostupne
41. return 0;
42. }
```

ZDROJ: vlastné spracovanie

2.2 Insertsort

Ďalšou priamou metódou je triedenie priamym vkladaním (Insertsort). V tejto metóde sa prvky poľa rozdelia na dve skupiny (postupnosti): zdrojovú a cieľovú. Prvky sa vyberajú zo zdrojovej a ukladajú sa do cieľovej skupiny. Najskôr sa vyberie druhý prvok v poradí zo zdrojovej postupnosti a vloží sa na príslušné miesto v cieľovej postupnosti. Následne sa rovnakým spôsobom vyberie a vloží tretí, štvrtý až posledný prvok a pole je usporiadané. Daný prvok sa na správne miesto v cieľovej postupnosti dostane tak, že ho porovnáme so susedným prvkom vľavo. Ak je susedný prvok väčší, tak ho posunieme vpravo o jedno miesto. Proces hľadania správneho miesta sa skončí v dvoch prípadoch:

1. Susedný prvok je menší ako prvok, ktorému hľadáme miesto.
2. Dostaneme sa na koniec cieľovej postupnosti. (Wirth, 1975)

Princíp triedenia priamym vkladaním môže byť zapísaný v pseudokóde ako je znázornené na obrázku 2.7.

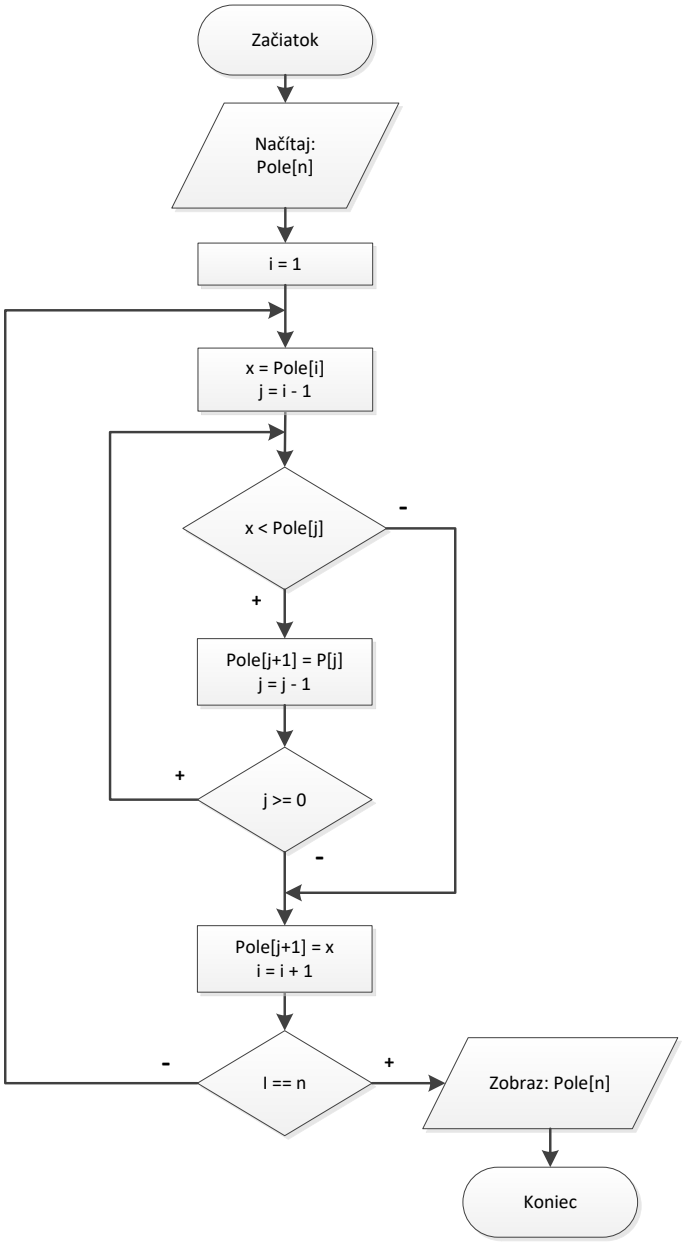
Obrázok 2.7 Pseudokód - Insertsort

```
Vstup: Pole P[n]
1. Pre i = 1,...n-1:
2.         x ← P[i]
3.         j ← i-1
4.         Pokiaľ je x < P[j] a zároveň je j >= 0:
5.                 P[j+1] ← P[j]
6.                 j ← j-1
7.         P[j+1] ← x
Výstup: Usporiadané pole P
```

ZDROJ: vlastné spracovanie

Z vyššie napísaného pseudokódu vytvoríme vývojový diagram, ktorý je zobrazený na obrázku 2.8.

Obrázok 2.8 Vývojový diagram – Insertsort



ZDROJ: vlastné spracovanie

Postup triedenia poľa priamym vkladaním vidíme v tabuľke 2.6. Červenou sú zvýraznené prvky, ktorým aktuálne hľadáme miesto a modrou sú zvýraznené prvky, ktoré sa dostali na vhodnú pozíciu. Z tabuľky tiež vidno, že triedenie priamym vkladaním patrí k stabilným metódam triedenia, pretože je zachované relatívne usporiadanie prvkov s rovnakou hodnotu. Zdrojový kód s príslušným popisom pre triedenie Insertsort sa nachádza na obrázku 2.9.

Tabuľka 2.6 Triedenie priamym vkladaním

6	**2**	4	8	8	3	1	3	9	5
2	6	**4**	8	8	3	1	3	9	5
2	**4**	6	**8**	8	3	1	3	9	5
2	4	6	**8**	**8**	3	1	3	9	5
2	4	6	8	**8**	**3**	1	3	9	5
2	**3**	4	6	8	8	**1**	3	9	5
1	2	3	4	6	8	8	**3**	9	5
1	2	3	**3**	4	6	8	8	**9**	5
1	2	3	3	4	6	8	8	**9**	**5**
1	2	3	3	4	**5**	6	8	8	9

ZDROJ: vlastné spracovanie

Obrázok 2.9 Algoritmus Insertsort v jazyku C

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.  int Pole[100],n,i,j,x;                  //deklarácia poľa a premenných
5.  printf("Zadaj velkost pola:\n");
6.  scanf("%d", &n);                        //načítanie veľkosti poľa do premennej n
7.  printf("Zadaj prvky pola[%d]:\n", n);
8.  for(i = 0; i < n; i++)
9.          scanf("%d", &Pole[i]);          //načítanie prvkov poľa
10. for(i = 1; i < n; i++)
11. {
12.         x = Pole[i];                    //uloženie hodnoty i-tého prvku poľa do x
13.         j = i - 1;
14.         while(x < Pole[j] && j >= 0)
15.         {
16.                 Pole[j+1] = Pole[j];    //posunutie vpravo prvkov, ktoré majú väčšiu hodnotu ako x
17.                 j = j - 1;
18.         }
19.         Pole[j+1] = x;                  //uloženie hodnoty x na správne miesto
20. }
21. printf("\nUsporiadane pole:\n");
22. for (i = 0; i < n; i++)
23.         printf("%d  ",Pole[i]);         //výpis usporiadaného poľa vzostupne
24. return 0:
```

ZDROJ: vlastné spracovanie

2.3 Selectsort

Základným krokom triedenia priamym výberom je vyberanie prvku s najmenšou hodnotou a jeho umiestnenie na začiatok poľa. Pole si môžeme rozdeliť na dve časti. Jedna je na začiatku poľa a vkladáme do nej nájdené najmenšie prvky. Druhú časť tvoria ostatné prvky poľa, ktoré ešte nie sú usporiadané. Po každom nájdení a uložení najmenšieho prvku sa neusporiadaná časť poľa zmenší o 1 a usporiadaná časť sa zväčší o 1. Týmto sa docieli, že už usporiadaný prvok sa ďalej neporovnáva. Najmenší prvok z neusporiadanej časti sa vloží vždy na koniec usporiadanej časti. (Mareš, a iní, 2017)

Metóda triedenia priamym výberom je v podstate založená na troch krokoch:

1. Vybrať najmenší prvok z poľa s n prvkami.
2. Vymeniť ho s prvým prvkom poľa.
3. Opakovať kroky 1 a 2 s ostávajúcimi n-1 prvkami, n-2 prvkami, až nám zostane posledný prvok, ktorý bude najväčší. (Wirth, 1975)

Metóda priamym výberom je v istom pohľade opakom k metóde priameho vkladania. (Wirth, 1975)

Tabuľka 2.7 Triedenie priamym výberom

6	2	4	8	8	3	**1**	3	9	5
1	**2**	4	8	8	3	6	3	9	5
1	**2**	4	8	8	**3**	6	3	9	5
1	**2**	**3**	8	8	4	6	**3**	9	5
1	**2**	**3**	**3**	8	**4**	6	8	9	5
1	**2**	**3**	**3**	**4**	8	6	8	9	**5**
1	**2**	**3**	**3**	**4**	**5**	**6**	8	9	8
1	**2**	**3**	**3**	**4**	**5**	**6**	**8**	9	8
1	**2**	**3**	**3**	**4**	**5**	**6**	**8**	9	**8**
1	**2**	**3**	**3**	**4**	**5**	**6**	**8**	**8**	9

ZDROJ: vlastné spracovanie

Princíp usporadúvania priamym výberom vidíme v tabuľke 2.7. Najskôr sa vyberie najmenší prvok (vyznačený červenou) a následne sa vymení s prvým prvkom z neusporiadanej časti poľa. Takto uložený najmenší prvok je zvýraznený modrou.

V tabuľke tiež vidíme zachované relatívne usporiadanie prvkov s rovnakou hodnotou. Môžeme teda povedať, že metóda triedenia priamym výberom je stabilná.

Pre lepšie pochopenie princípu triedenia priamym výberom je vhodný obrázok 2.10, na ktorom je znázornený pseudokód. Do premennej m sa ukladá index aktuálne najmenšieho prvku.

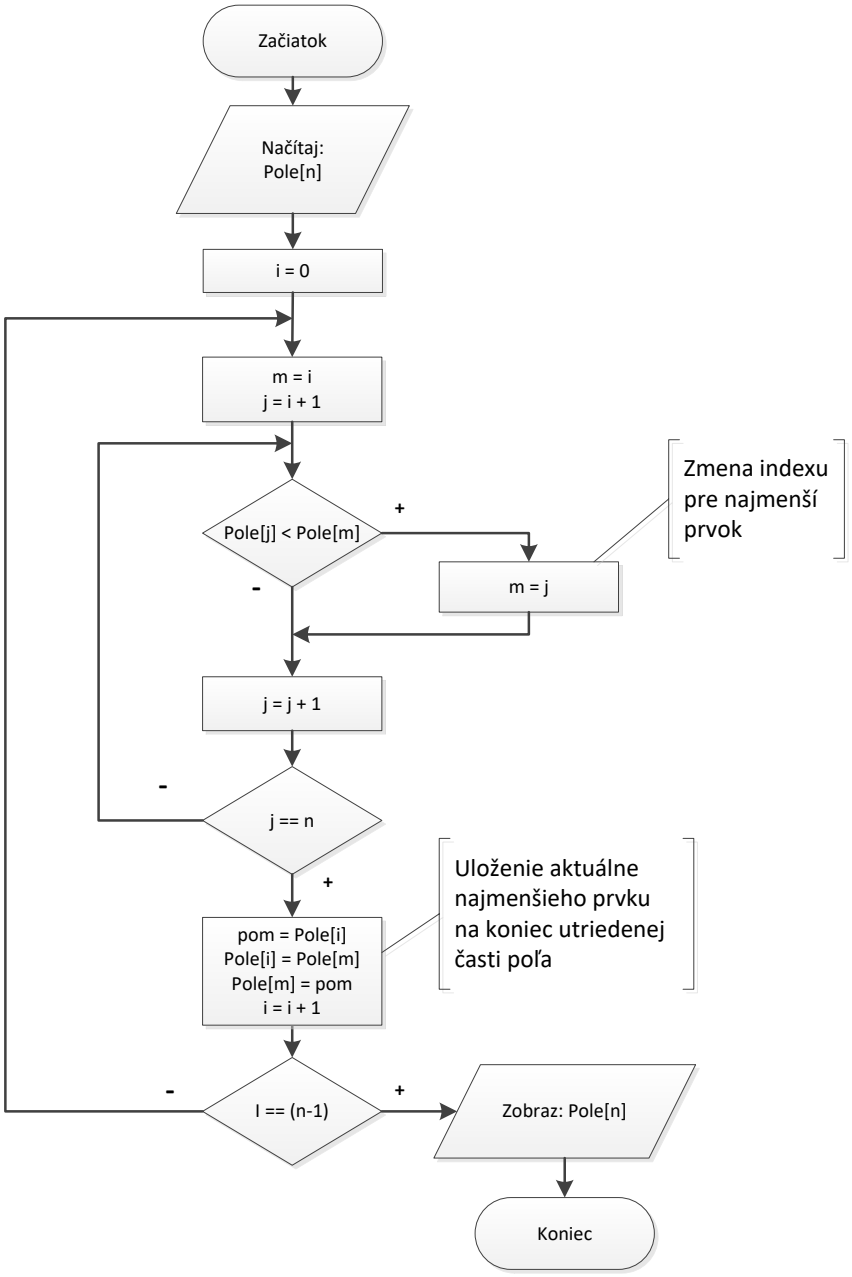
Obrázok 2.10 Pseudokód - Selectsort

```
Vstup: Pole P[n]
1. Pre i = 0,...,n-2:
2.         m ← i
3.         Pre j = i+1,...,n-1:
4.                 Ak je P[j] < P[m]:
5.                         m ← j
6.         Prehoď prvky P[i] a P[m]
Výstup: Usporiadané pole P
```

ZDROJ: vlastné spracovanie podľa (Mareš, a iní, 2017)

Pomocou pseudokódu sa vytvorí vývojový diagram – obrázok 2.11, v ktorom sú popísané hlavné bloky typické pre algoritmus Selectsort. Hlavné bloky sú ukladanie indexu najmenšieho prvku a uloženie najmenšieho prvku na koniec utriedenej časti poľa. Následne sa z vývojového diagramu napíše zdrojový kód v jazyku C (obrázok 2.12).

Obrázok 2.11 Vývojový diagram – Selectsort



ZDROJ: vlastné spracovanie

Obrázok 2.12 Algoritmus Selectsort v jazyku C

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.  int Pole[100],n,i,j,m,pom;                  //deklaracia pola a premennych
5.  printf("Zadaj velkost pola:\n");
6.  scanf("%d", &n);                            //nacitanie velkosti pola do premennej n
7.  printf("Zadaj prvky pola[%d]:\n", n);
8.  for(i = 0; i < n; i++)
9.          scanf("%d", &Pole[i]);              //nacitanie prvkov pola
10. for(i = 0; i < n-1; i++)
11. {
12.         m = i;
13.         for(j = i+1; j < n; j++)
14.         {
15.                 if(Pole[j] < Pole[m])
16.                         m = j;
17.         }
18.         pom = Pole[i];
19.         Pole[i] = Pole[m];
20.         Pole[m] = pom;
21. }
22. printf("\nUsporiadane pole:\n");
23. for (i = 0; i < n; i++)
24.         printf("%d  ",Pole[i]);              //vypis usporiadaneho pola vzostupne
25. return 0;
26. }
```

ZDROJ: vlastné spracovanie

2.4 Funkcia sínus

S funkciou sínus sme sa už stretli asi všetci. Používa sa pomerne často pri rôznych výpočtoch, či už v matematike, fyzike ale aj v elektrických obvodoch. Málokto sa ale zamýšľa nad tým, ako sa hodnota sínusu konkrétneho uhla vypočíta. Väčšina je zvyknutá použiť funkciu sínus v kalkulačke, kedy zadajú uhol v stupňoch alebo radiánoch a vypíše sa im výsledok. Málokto vie, že sínus sa v skutočnosti počíta pomocou Taylorovho radu. Výpočet sínusu uhla x v radiánoch sa realizuje nasledovne:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots$$

Presnosť výsledku závisí od veľkosti polynómu - čím viac členov rad obsahuje, tým je presnosť väčšia a tým pádom aj presnejší výsledok.

V programovaní v jazyku C sa používa preddefinovaná funkcia sin(x), ktorá vráti hodnotu sínusu zadaného uhla x v radiánoch. Táto funkcia je obsiahnutá v knižnici math.h, ktorá v sebe zahŕňa množstvo ďalších matematických funkcií ako napríklad goniometrické, exponenciálne alebo logaritmické. (Programiz, 2020)

Princíp riešenia funkcie sínus cez algoritmus spočíva v pripočítavaní alebo odpočítavaní čoraz menšej hodnoty. Pre výpočet sínusu si zvolíme hodnotu uhla x a požadovanú presnosť, ktorú má mať výsledok. Následne sa budú v cykle vypočítavať jednotlivé členy Taylorovho radu a tie sa budú pripočítavať k výsledku. Hneď ako bude hodnota posledného člena radu menšia ako je zadaná presnosť, cyklus sa ukončí a zobrazí sa výsledok. Takto popísaný princíp výpočtu si môžeme všimnúť na obrázku 2.13, kde je zobrazený pseudokód pre výpočet hodnoty sínusu. Premenná t reprezentuje nový člen v Taylorovom rade a do premennej výsledok sa priebežne ukladá vypočítaná hodnota sínusu. Nakoniec bude v premennej výsledok uložený konečný výsledok výpočtu. (PŠENČÍKOVÁ, 2009)

Obrázok 2.13 Pseudokód - sínus

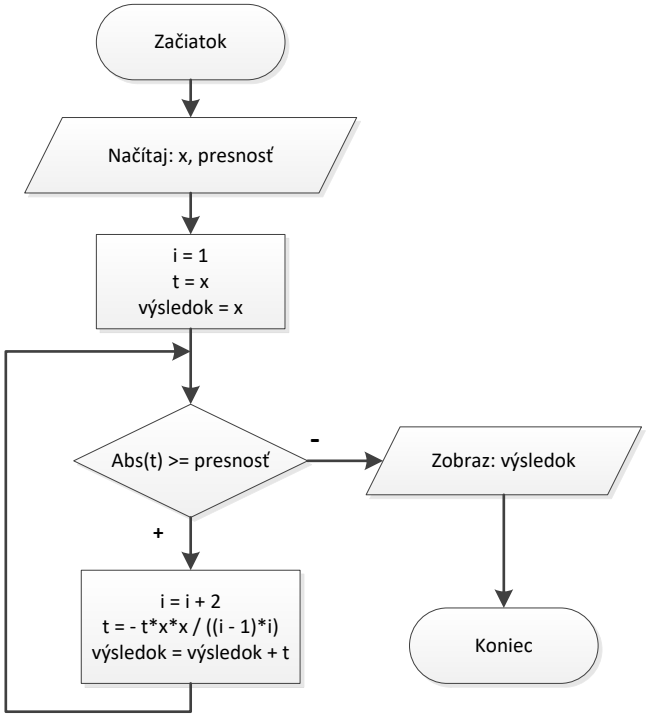
Vstup: uhol x, presnosť
1. i ← 1
2. t ← x
3. výsledok ← x
4. Pokiaľ je abs(t) >= presnosť:
5. i ← i + 2
6. t ← t*x*x / ((i - 1)*i)
7. výsledok ← výsledok + t

Výstup: Výsledok

ZDROJ: vlastné spracovanie podľa

Pre lepšiu interpretáciu vytvoríme z pseudokódu vývojový diagram, ktorý je zobrazený na obrázku 2.14.

Obrázok 2.14 Vývojový diagram - sínus



ZDROJ: vlastné spracovanie podľa (PŠENČÍKOVÁ, 2009)

Z vývojového diagramu vytvoríme zdrojový kód v jazyku C (obrázok 2.15). Presnosť si zvolíme podľa toho, aký presný chceme výsledok. Treba však dať pozor na to, že hodnota presnosti neudáva rozdiel medzi výpočtom sínusu cez navrhnutý algoritmus a výpočtom cez funkciu sin() z knižnice math.h. Výpočet sínusu cez algoritmus je dostatočne presný už pri polynóme 9. stupňa.

Obrázok 2.15 Algoritmus sínus v jazyku C

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main()
4.  {
5.  double x,presnost,vysledok,t;                    //deklaracia premennych typu double
6.  int i = 1;                                       //deklaracia premennej typu int
7.  printf("Zadaj uhol v radianoch:\n");
8.  scanf("%lf", &x);
9.  printf("Zadaj presnost:\n");
10. scanf("%lf", &presnost);
11. t = x;
12. vysledok = x;
13. while(fabs(t)>= presnost)                        //ak je absolutna hodnota posledneho
    prirastku mensia ako presnost, cyklus sa nevykona
14. {
15. i = i + 2;
16. t = -t*x*x/((i-1)*i);
17. vysledok = vysledok + t;
18. }
19. printf("sin(%lf) = %lf\n", x, vysledok);
20. return 0;
21. }
```

ZDROJ: vlastné spracovanie

3 POPIS REALIZÁCIE VYBRANÝCH ALGORITMOV

V tejto kapitole uvediem popis algoritmov z druhej kapitoly v prehľadovej forme pomocou tabuľky. Algoritmy sú realizované v pseudokóde, vývojovom diagrame, tabuľkách s prechodmi poľom, jazyku C vo vývojovom prostredí CodeBlocks, ktorá slúži ako platforma pre počítač a v prostredí Code Composer Studio (CCS), ktoré sa používa pre platformu mikrokontroléra MSP430 od firmy Texas Instrument. Prostredie CodeBlocks sa používa vo výučbe na Fakulte elektrotechniky a informačných technológií a s prostredím CCS sa pracuje na Katedre mechatroniky a elektroniky na univerzite v Žiline.

3.1 Prehľad realizovaných algoritmov

Prehľadová tabuľka 3.1 slúži na rýchle zorientovanie sa v jednotlivých algoritmoch a taktiež na jednoduché vyhľadávanie jednotlivých realizácií konkrétnych algoritmov v práci. V každom riadku tabuľky je uvedený názov algoritmu, číslo podkapitoly a následne obrázky a tabuľky s príslušným číslom strany, kde sa daná realizácia algoritmu nachádza. Projekt pre platformu MSP430 sa nachádza v elektronických prílohách.

Tabuľka 3.1 Prehľadová tabuľka realizovaných algoritmov a ich popisov

Číslo riadku	Názov algoritmu	Číslo podkapitoly	Pseudokód	Vývojový diagram	Prechod algoritmom v cykloch	Zdrojový kód v jazyku C	Projekt pre MSP430
1.	Bubblesort	2.1	Obrázok 2.1 s. 29	Obrázok 2.2 s. 31	Tabuľka 2.1 s. 29 Tabuľka 2.2 s. 30	Obrázok 2.3 s. 32	Elektronická príloha č.1
2.	Shakesort	2.1.1	Obrázok 2.4 s. 33	Obrázok 2.5 s. 34	Tabuľka 2.3 s. 35 Tabuľka 2.4 s. 35 Tabuľka 2.5 s. 36	Obrázok 2.6 s. 36	Elektronická príloha č.2
3.	Insertsort	2.2	Obrázok 2.7 s. 38	Obrázok 2.8 s. 39	Tabuľka 2.6 s. 40	Obrázok 2.9 s. 40	Elektronická príloha č.3
4	Selectsort	2.3	Obrázok 2.10 s. 42	Obrázok 2.11 s. 43	Tabuľka 2.7 s. 41	Obrázok 2.12 s. 44	Elektronická príloha č.4
5.	Funkcia sínus	2.4	Obrázok 2.13 s. 45	Obrázok 2.14 s. 46		Obrázok 2.15 s. 47	Elektronická príloha č.5

ZDROJ: vlastné spracovanie

Záver

Cieľom mojej bakalárskej práce bolo vytvoriť vhodné algoritmické úlohy pre použitie na platforme MSP430, ktorá sa používa vo výučbe na Katedre mechatroniky a elektroniky na Fakulte elektrotechniky a informačných technológií v Žiline.

Prvý čiastkový cieľ práce teoretické vymedzenie pojmov bol realizovaný v prvej kapitole, kde som zadefinoval pojmy dôležité na pochopenie danej problematiky. Druhý čiastkový cieľ prehľad vybraných algoritmov so zameraním na triediace a numerické algoritmy sa nachádza v druhej kapitole, kde je popísaný princíp jednotlivých algoritmov. Tretí čiastkový cieľ návrh a realizácia vybraných algoritmov na mikroprocesorovej platforme MSP430 sa nachádza v elektronických prílohách v podobe projektov pre vývojové prostredie Code Composer Studio, ktoré si môže používateľ naimportovať do vývojového prostredia na svojom počítači. Elektronické prílohy k bakalárskej práci sa nachádzajú na priloženom CD a taktiež je ich možné stiahnuť z linku na Google Drive:

https://drive.google.com/file/d/1V5begd1S_t2wFyR9wocS7sze5Giw1sZ7/view?usp=sharing

Práca má poslúžiť ako pomôcka tak vyučujúcemu ako aj samotným študentom predmetu Mikroprocesorové systémy. V práci som mohol spomenúť viac numerických algoritmov, ale z hľadiska neprekročenia rozsahu práce to nebolo možné. Na záver sa domnievam, že som hlavný cieľ a aj čiastkové ciele splnil.

Zoznam použitej literatúry

ABAS, M. 2020. Algoritmy. *www.iring.sk.* [Online] 2020. [Dátum: 28. 4 2020.] http://www.iring.sk/downloads/algoritmy.pdf.

DRLÍK, P. 2020. Turbo PASCAL 1. *www.enigma.sk.* [Online] 2020. [Dátum: 23. 4 2020.] www.enigma.sk/download/476.

Guru99. 2020. What is C Programming Language? Basics, Introduction and History. *https://www.guru99.com/.* [Online] 2020. [Dátum: 23. Apríl 2020.] https://www.guru99.com/c-programming-language.html.

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

Mareš, M. a Valla, T. 2017. *Průvodce labyrintem algoritmů.* Praha : CZ.NIC, z. s. p. o., 2017. ISBN 978-80-88168-22-5.

Programiz. 2020. C sin() - C Standard Library. *https://www.programiz.com/.* [Online] 2020. [Dátum: 27. 5 2020.] https://www.programiz.com/c-programming/library-function/math.h/sin.

PŠENČÍKOVÁ, J. 2009. *Algoritmizace.* Kralice na Hané : Computer Media s.r.o., 2009. ISBN 978-80-7402-034-6.

Skalka, J., a iní. 2007. *Algoritmizácia a úvod do programovania.* Nitra : UKF v Nitre, 2007. ISBN 978-80-8094-217-5.

Wirth, N. 1975. *Algoritmy a štruktúry údajov.* [prekl.] Pavol Fischer. Bratislava : Vydavateľstvo technickej a ekonomickej literatúry, 1975. ISBN 80-05-00153-3.