TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksei Karagodov

SORTIMISALGORITMIDE ANALÜÜS EKSTREMUMITE LEIDMISEKS MIKROKONTROLLERI RAKENDUSTES

Bakalaureusetöö

Juhendaja: Vladimir Viies

Dotsent

Tallinn 2021

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Aleksei Karagodov

12.05.2021

Annotatsioon

Selle lõputöö eesmärk on analüüsida sorteerimisalgoritmi, et vastata küsimusele, milline sorteerimisalgoritme sobib mikrokontrollerile kõige paremini, et leida suurimad ja väiksemad arvandmed andmete loendis.

Antud töö peaks lahendama mikrokontrollerite jaoks tõhusama sortimisalgoritmi valimise probleemi, analüüsides sorteerimisalgoritme kasutades C ja C++ keeli.

Vastavalt püstitatud eesmärgile lisas autor 2 alamülesanet. Esimene alamülesanne on kaaluda sortimisalgoritme ja eemaldada kõige aeglasemad sorteerimisalgoritmid analüüsitakse algoritmide loendist. Teine alamülesanne on valida mikrokontroller.

Lõputöös analüüsitakse sorteerimisalgoritme teoreetiliste andmete põhjal ning analüüsitakse ka mikrokontrollereid. Pärast seda viiakse läbi mikrokontrolleri sorteerimisalgoritmide praktiline analüüs. Lõputöös kirjeldatakse ja selgitatakse kasutatud sorteerimisalgoritmide tähendust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 43 leheküljel, 3 peatükki, 32 joonist, 6 tabelit.

Abstract

Analysis of Sorting Algorithms for Finding Extremum in Microcontroller Applications

The aim of this thesis is to analyze a sorting algorithms to answer the question of which sorting algorithms are best suited for a microcontroller to find the largest and smallest numbers in the data list.

This thesis should solve the problem of choosing a more efficient sorting algorithm for microcontrollers by analysing sorting algorithms using C and C ++ programming languages.

According to the set goal, the author added 2 subtasks. The first subtask is to consider the sorting algorithms and remove the slowest sorting algorithms from the list of analysed algorithms.

Another subtask is to select a microcontroller, that has:

1. more than 100 different possible unique commands in the processor to efficiently execute the sorting algorithms;
2. such processor speed to sort the data quickly, but at the same time the processor speed is not such that it can sort the data "instantly" (between 10-30 MHz);
3. enough flash memory to load the program with a sorting algorithm (more than 10kB) and enough dynamic memory (RAM) to create an array of 750 elements, each element being 2 bytes in size. (RAM 2kB and higher).

In the thesis, sorting algorithms will be analysed on theoretical data, and microcontrollers will also be analysed. After that, a practical analysis of the sorting algorithms on the microcontroller will be carried out. Also, in the thesis the meaning of the used sorting algorithms will be described and explained.

The thesis is in estonian and contains 43 pages of text, 3 chapters, 32 figures, 6 tables.

Lühendite ja mõistete sõnastik

kB	*Kilobyte*
I/O	*Input/Output*, Sisend/Väljund
CPU	*Central Processing Unit*, Keskseade
ADC	*Analog-to-Digital Converter*, Analoog-digitaalmuundur; kasutatakse elektroonikas
DAC	*Digital-to-Analog Converter*, Analoog-digitaalmuundur; kasutatakse elektroonikas
ROM	*Read-Only Memory*, püsimälu; Teisisõnu mälu, mida saab muuta ainult teatud tingimustel.
RAM	*Random Access Memory*, Muutmälu; Ajutine mälu protsessorile, kuhu ta saab kirjutada ja kust ka andmeid lugeda.
SRAM	*Static Random Access Memory*, Staatiline muutmälu; Juhuslikus järjekorras salvestatav või loetav mäluseade, milles sisalduvad andmed ei vaja nende säilitamiseks perioodilist uuendamist, kuid see vajab toitepinget, et andmeid säilitada.
ISA	*Instruction Set Architecture*, Käsustik; Käsustik määratleb protsessoris käskude arhitektuuri ja paigutuse
RISC	*Reduced Instruction Set Computer*, Kärbitud käsustikuga arvuti; Üks võimalikest käsustikkudest. Igale protsessori käsule eraldatakse eraldi mälu.
SMD-komponent	*Surface-Mount Device component*, pinnale paigaldatav seade komponent; Sarnaseid komponente (need võivad olla takistid, kondensaatorid jne) kasutatakse trükkplaatidel.
Tsükkel (Programmeerimisel)	Mõne käsud, mida korratakse teatud arv kordi.
For-tsükkel	Tsükkel, mis käivitatakse spetsiaalse käsu "for" abil.

Sisukord

Jooniste loetelu

Tabelite loetelu

Sissejuhatus

Igal aastal kasvab pidevalt selliste seadmete hulk, mille põhjal saab mikrokontroller nagu nutitelefonid, nutikellad ja erinevad andurid. Nende arvutusvõimsus suureneb ja nende suurus muutub kompaktsemaks. Nii et näiteks statistika järgi kasutas 2012. aastal nutitelefoni 1,06 miljardit inimest ja 2020. aastal 3,6 miljardit nutitelefoni [1]. Need seadmed peavad töötama piisavalt kiiresti ja kõik selliste seadmete mikrokontrollerites olevad arvutusprotsessid peavad toimuma võimalikult kiiresti ja tõhusalt, kasutades kõige vähem ressursse. Selleks on selliste seadmete mikrokontrollerid programmeeritud nii, et need eesmärgid saavutatakse.

Üks vajalik operatsioone on algoritmide sortimine, mis sorteerib andmeid õiges järjekorras. Sortimisalgoritme on suur hulk, mille abil on võimalik andmeid sortida. Igal neist sortimisalgoritmidest on eeliseid ja puudusi.

Üheks ülesandeks on vaja kogutud andmeid sortida. Kui selline vajadus tekib mikrokontrollerites, näiteks andurites, siis tuleb selline algoritm täita võimalikult kiiresti ja tõhusalt.

Seadis selle töö autor endale eesmärgi analüüsida sorteerimisalgoritmi, et vastata küsimusele, milline sorteerimisalgoritme sobib mikrokontrollerile kõige paremini, et leida suurimad ja väiksemad arvandmed andmete loendis.

Antud töö peaks lahendama mikrokontrollerite jaoks tõhusama sortimisalgoritmi valimise probleemi, analüüsides sorteerimisalgoritme kasutades C ja C++ keeli.

Vastavalt püstitatud eesmärgile lisas autor 2 alamülesanet. Esimene alamülesanne on kaaluda sortimisalgoritme ja eemaldada kõige aeglasemad sorteerimisalgoritmid analüüsitakse algoritmide loendist.

Teine alamülesanne on valida mikrokontroller, millel:

1. on protsessoris erinevate võimalike unikaalsete käskude arv oleks rohkem kui 100, et sorteerimisalgoritme tõhusalt täita;

2. protsessori kiirus oleks selline, et kiiresti andmeid sortida, kuid samal ajal ei oleks protsessori kiirus selline, et sortida andmeid "koheselt" (vahemikus 10-30 MHz);
3. oleks piisavalt välkmälu programmi sortimisalgoritmiga laadimiseks (rohkem, kui 10kB) ja piisavalt dünaamilise mälu (RAM), et luua 750 elementi massiiv, millest igaüks element on 2 baiti suurusega. (RAM 2kB ja suurem).

Seoses püstitatud eesmärgi ja alamülesannetega jaguneb töö 3 osaks. Töö esimeses kaalub autor sorteerimisalgoritmi ja teeb neist lühikese analüüsi, et valida töö jaoks sortimisalgoritmid, mille sortimise kiirus on piisavalt suur. Teises osas kaalub autor mikrokontrollereid ja teeb neist teise alamülesande täitmiseks lühikese analüüsi. Viimases osas käsitleb autor valitud sortimisalgoritmide otsest praktilist analüüsi mikrokontrolleris, et saavutada selle töö eesmärk.

1 Sortimisalgoritmid

Selles peatükis käsitletakse mõningaid tuntud sorteerimisalgoritme ja viiakse läbi nende teoreetiline analüüs, et valida tööülesannete täitmiseks kõige sobivamad sortimisalgoritmid.

1.1 «Big O notation»

Peamine viis erinevate sortimisalgoritmide võrdlemiseks on kasutada matemaatilist tähistust «Big O notation».

Tegelikult on algoritmide sorteerimise matemaatilise tähistuse "Big O notation" peamine ülesanne kirjeldada saadud algoritmi sõltuvust mõnest muust muutujast.

Oletame, et on olemas funktsioon, mis korrutab 2 arvu ja tagastab seejärel nende korrutise. (vt Joonis 1)

```
int mul(int a, int b) {
	return a * b;
}
```

Joonis 1. Näide O(1) funktsiooni sõltuvusi

Kuna sortimisalgoritmid käsitlevad suuremat andmemahtu, võetakse alati arvesse algandmete suurust. Kuid sel juhul, kui oleks vaja hinnata ühe sellise funktsiooni täitmiseks kuluvat aega, võttes arvesse andmesuurust n, on selge, et sellel funktsioonil pole midagi pistmist andmete suurusega ja see töötab alati sama palju aega. Seega on võimalik järeldada, et see funktsioon sõltub ülaltoodud tingimusest O(1) [2].

Oletame nüüd, et on olemas funktsioon, mis kasutab andmete printimiseks ühte silmust teise sees. (vt Joonis 2)

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

Joonis 2. Näide O(n^2) funktsiooni sõltuvusi

Sellisel juhul on olemas 2 for-tsüklit, millest igaüks sõltub andmete suurusest. Pealegi on üks tsüklitest teise sees. Seega, kui meie andmete suurus on n elementi, siis esimene tsükkel kutsub teist tsüklist täpselt n korda ja omakorda teine tsükkel prindib ise täpselt n andmeid. Seega printib funktsioon täpselt n^2 andmeid või prinditavate andmete hulk sõltub selle suurusest suurusega O(n^2) [2].

Oletame, et lisasime oma viimasele funktsioonile veel ühe silmuse. (vt Joonis 3)

```
void printData(int *arr, size_t n) {
	for (int i = 0; i < n) printf("New data\n");
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++) printf("%d\n", arr[i]);
}
```

Joonis 3. Näide O(n + n^2) funktsiooni sõltuvusi

Sel juhul lisati funktsioonile lisaks kahele eelmisele tsüklile veel üks tsükkel, mis sõltub andmete suurusest. Sellisel juhul võib prinditud andmete hulga sõltuvust nende suurusest kirjeldada kui O(n + n^2). Sorteerimisalgoritmide hindamisel jääb tavaliselt siiski kõige olulisem sõltuvus [2]. Teisisõnu, kuna $n < n^2$, kui andmete suurus on üsna suur (oletame, et n = 700), siis on n^2 palju suurem kui n (700^2 versus 700, väga suur arv). Nii et ülaloleval funktsioonil oleks andmete suuruse sõltuvus O(n^2).

1.2 Algoritmi hindamiskriteeriumid

Sorteerimisalgoritmide hindamisel kasutatakse tavaliselt kolme mõõdet: aja keerukus (time complexity), ruumi keerukus (space complexity) ja stabiilsus (stability). Tulevikus kasutatakse just neid hindamiskriteeriume sortimisalgoritmide efektiivsuse hindamiseks. (vt jaotis 1.4)

1.2.1 Aja keerukus

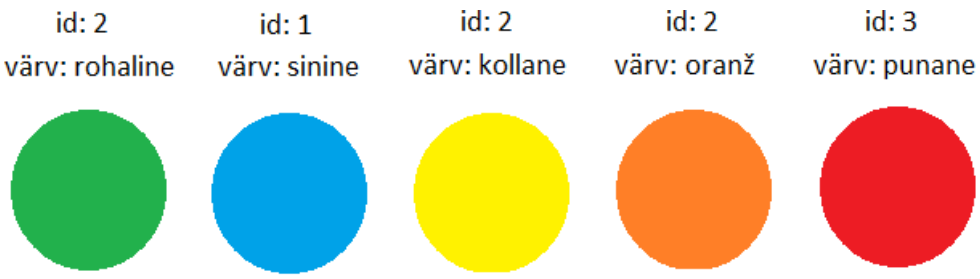
Esimene parameeter, time complexity, peegeldab aega, mis kulub sortimisalgoritmil n-nda andmemahu sorteerimiseks. Oletame, et on olemas sortimisalgoritm X, mis sorteerib kõik andmed O(n) ajaga, ja on olemas sortimisalgoritm Y, mis sorteerib kõik andmed O (n^2) ajas. Seega on sortimisalgoritm X kiirem kui sortimisalgoritm Y, kuna $O(n) < O(n^2)$. Samuti on algoritmi kiiruse hindamisel parim juhtum (best-case) (juhtum, kus andmed on juba sorteeritud), halvim(worst-case) (juhtum, kus andmed on paigutatud nii, et selle võtmiseks kulub kõige rohkem aega) võetakse arvesse andmete sortimist) ja andmete sortimise keskmist juhtumit(average time complexity) [2] [3]. Oletame, et on olemas sorteerimisalgoritm X, mis sorteerib andmed parimal juhul O (n) ajaga, keskmiselt ja halvimal juhul $O(n^2)$ ajaga, ning on olemas sortimisalgoritm Y, mis sorteerib andmeid igal juhul O(n*log (n)) ajaga. Sel juhul pole enam üheselt mõistetav öelda, et sortimisalgoritm Y on alati parem kui sortimisalgoritm X.

1.2.2 Ruumi keerukus

Teine parameeter - space complexity - kajastab mõne sortimisalgoritmi jaoks vajaliku täiendava ajutise salvestusruumi hulka. Seega, kui sortimisalgoritm kasutab O(1) mälumahtu, siis ei sõltu sisendandmete hulgast mingi püsiv mälumaht. Kuid kui sorteerimisalgoritm kasutab O(n) lisamälu, siis see tähendab, et andmete sortimise protsessis kasutatakse lisaks sama palju mälu, mida kasutatakse n-elementide jaoks [4]. See tähendab, et kui peate sorteerima 10 üksust, millest igaüks võtab mälus 4 baiti, siis selle sortimisalgoritmi abil luuakse ajutiselt 10 täiendavat üksust, millest igaüks võtab mälus 4 baiti. Ruumi keerukuse hindamisel, samuti aja keerukuse hindamisel kasutatakse parimat, halvimat ja keskmist statistilist juhtumit.

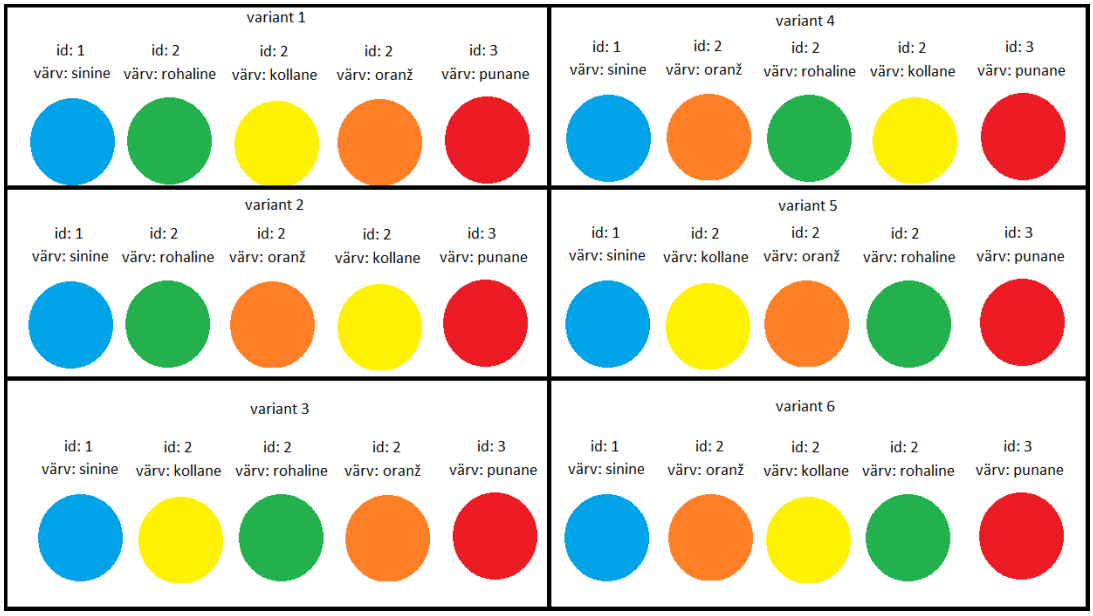
1.2.3 Stabiilsus

Viimane parameeter, stabiilsus, peegeldab seda, kas sortimisalgoritm järgib andmete järjekorda, nagu see oli andmete sortimisel [5]. See tähendab, et on olemas 5 elementi, mida on vaja sortida nende ID järgi. Samuti igal elemendil on olemas täiendav “värv” element. (vt Joonis 4)



Joonis 4. Sorteerimata andmed

Kui kasutatakse ebastabiilset sorteerimisalgoritmi, siis massiivi elementide järjestus ei oma tähtsust ja selle tulemusena võivad sorteeritud andmed erineda. See tähendab, et selle tulemusena võivad osutuda samade sorteeritud andmete erinevad variatsioonid. Näiteks kui algandmetel oli pildile vastav järjekord, st "2 rohaline - 1 sinine - 2 kollane - 2 oranž - 3 punane", siis tulenevalt sellest, ebastabiilse sorteerimise algoritmi juurutamisel saab see sorteeritud andmete jaoks 6 erinevat võimalust. (vt Joonis 5)



Joonis 5. Kõik võimalikud valikud andmete sorteerimiseks ebastabiilse sortimisalgoritmi abil

Kui kasutatakse stabiilset sorteerimisalgoritmi, siis selle tulemusel sorteeritakse andmed samas järjekorras, milles need asuvad enne andmete sortimise algust. See tähendab, et kui algandmetel oli järjestus vastavalt pildile (vt Joonis 4), siis selle tulemusel näevad sorditud andmed välja esimese võimalusena. (vt Joonis 5)

1.3 Erinevad sortimisalgoritmid

Selles peatükis käsitletakse erinevaid sorteerimisalgoritme. Selguse huvides toimub sortimine andmemassiivis [3, 4, 6, 2, 1, 5].

1.3.1 Mullsortimine

Mullsortimine on kõige lihtsam ja intuitiivsem andmete sorteerimise algoritm.

Mullsortimine abil liigub “viit” samm-sammult massiivi esimesest elemendist viimasesse ning enne igat sammu võrdleb algoritm elementi, millel “viit” asub, massiivi järgmise elemendiga ja kui esimene number on suurem kui järgmine, siis vahetab algoritm esimese numbri teise numbriga ja pärast seda liigub "viit" järgmise elemendi juurde. Kui esimene number on väiksem kui järgmine number, liigub “viit” lihtsalt järgmise elemendi juurde. Kui "viit" liigub viimase elemendi juurde, naaseb see esimese juurde ja kordab eelnevalt kirjeldatud algoritmi massiivi numbrite permutatsiooniga. Kokku toimub selline protseduur nii mitu korda, kui massiivis on elemente (n), ainult üks vähem (n-1).

Tabel 1. Mullsortimise näide

	Esimese (6) numbri sorteerimine	**Teine (5) numbri sorteerimine**	**Kolmanda (3) numbri sorteerimine**	**Neljanda (4) numbri sorteerimine**	**Viimase (2) numbri sorteerimine**
i = 0	[***3***, **4**, 6, 2, 1, 5]	[***3***, **4**, 2, 1, 5, 6]	[***3***, **2**, 1, 4, 5, 6]	[***2***, **1**, 3, 4, 5, 6]	[***1***, **2**, 3, 4, 5, 6]
i = 1	[3, ***4***, **6**, 2, 1, 5]	[3, ***4***, **2**, 1, 5, 6]	[2, ***3***, **1**, 4, 5, 6]	[1, ***2***, **3**, 4, 5, 6]	[1, ***2***, **3**, 4, 5, 6]
i = 2	[3, 4, ***6***, **2**, 1, 5]	[3, 2, ***4***, **1**, 5, 6]	[2, 1, ***3***, **4**, 5, 6]	[1, 2, ***3***, **4**, 5, 6]	[1, 2, ***3***, **4**, 5, 6]
i = 3	[3, 4, 2, ***6***, **1**, 5]	[3, 2, 1, ***4***, **5**, 6]	[2, 1, 3, ***4***, **5**, 6]	[1, 2, 3, ***4***, **5**, 6]	[1, 2, 3, ***4***, **5**, 6]
i = 4	[3, 4, 2, 1, ***6***, **5**]	[3, 2, 1, 4, ***5***, **6**]	[2, 1, 3, 4, ***5***, **6**]	[1, 2, 3, 4, ***5***, **6**]	[1, 2, 3, 4, ***5***, **6**]
i = 5	[3, 4, 2, 1, 5, ***6***]	[3, 2, 1, 4, 5, ***6***]	[2, 1, 3, 4, 5, ***6***]	[1, 2, 3, 4, 5, ***6***]	[1, 2, 3, 4, 5, ***6***]

Kuna selle sorteerimise korral peab algoritm jõudma massiivi algusest lõpuni n korda, kus n on massiivi pikkus, ning samuti sorteerima iga üksiku numbri, millest massiivis on n tükki, siis ta on analüütiliselt saadud selle tulemusena, et algoritmi sorteerimiseks kuluvat aega saab väljendada kui $O(n^2)$.

Selle algoritmi juurutamisel ei kaasata täiendavaid parameetreid, mis sõltuksid massiivi suurusest ja ainus toimuv toiming on muutujate kohtade asendamine massiivis, seetõttu on sellise algoritmi ruumiline keerukus O(1).

Lisaks hõivavad aja keerukuse ja ruumi keerukuse parameetrid parimal, keskmisel ja halvimal juhul vastavalt $O(n^2)$ ja O (1) koha.

Tulenevalt asjaolust, et algoritm asendab numbreid ainult siis, kui esimene number on suurem kui teine, on selline algoritm stabiilne.

Mullsorteerimise algoritm on väga hõlpsasti arusaadav ning andmete sorteerimiskood ise on äärmiselt lihtne ja võtab vähem kui 10 rida.

```
void bubbleSort (int *arr, size_t length) {
	int temp;
	for (int i = 0; i < length; i++) {
		for (int j = 0; j < length - 1; j++) {
			if (arr[j] > arr[j + 1]) {
				temp = arr[j + 1];
				arr[j + 1] = arr[j];
				arr[j] = temp;
			}
		}
	}
}
```

Joonis 6. Mullsortimise algoritmi rakendamine

See kood (vt Joonis 6) rakendab mullsortimiini kõige lihtsamal tasemel. Kuid seda algoritmi saab optimeerida asjaoluga, et eelmise tsükli viimast elementi ei kontrollita teist korda (see tähendab, et j < length - i - 1). Samuti on võimalik kontrollida, kas esimese tsükli jooksul massiivi kahe elemendi asendamise tingimus oli täidetud, sest kui asendamist ei toimunud, siis tegu on parimaga juhuga, mille korral ajakeerukus võtab ainult O(n) aega.

```
void bubbleSort (int *arr, size_t length) {
	int temp, swap = 0;
	for (int i = 0; i < length; i++) {
		for (int j = 0; j < length - i - 1; j++) {
			if (arr[j] > arr[j + 1]) {
				temp = arr[j + 1];
				arr[j + 1] = arr[j];
				arr[j] = temp;
				swap = 1;
			}
		}
		if (!swap) break;
	}
}
```

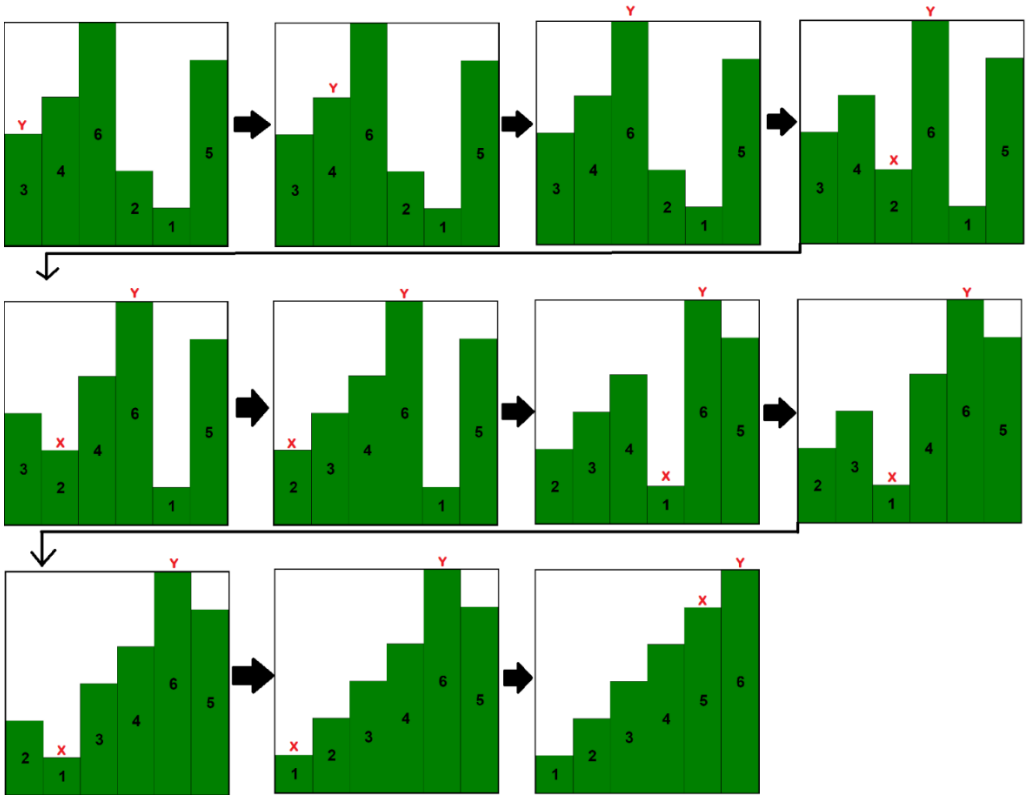
Joonis 7. Optimeeritud mullsotimine algoritm

See kood (vt Joonis 7) sisaldab kõiki võimalikke optimeerimisi, mis võivad koodi kiirendada.

1.3.2 Vahelepanemisega sortimine

Vahelepanemisega sortimine, nagu mullsortimine, on äärmiselt hõlpsasti mõistetav sorteerimisalgoritm.

Kasutades vahelepanemisega sortimine, algab “viit” esimesest elemendist ja selle tulemusena peab see samm-sammult massiivi lõpuni jõudma. Kui järgmisel elemendil on suurem väärtus kui eelmisel elemendil, liigub “viit” lihtsalt järgmise elemendi juurde. Kui praegusel elemendil (olgu selle elemendi nimi Y) on suurem väärtus kui järgmisel (olgu selle elemendi nimeks X), siis algoritm hakkab väiksemat elementi oma kohale sisestama. See tähendab, et algoritm säilitab Y-elemendi positsiooni ja hakkab astuma vastupidises suunas, samal ajal võrreldes X-elementidega, mis olid Y-elemendi taga. Kui on element, mille väärtus on väiksem kui X-element, siis sisestab algoritm X elemendi selle elemendi ette. Vastasel juhul läheb algoritm massiivi kõige esimesse positsiooni ja lisab elemendi kõige esimesse positsiooni. Pärast seda, kui algoritm on massiivi alguse ja Y vahele lisanud elemendi X, pöördub algoritm tagasi elemendi Y juurde ja jätkab algse ülesande täitmist kuni massiivi viimase elemendini.



Joonis 8. Vahelepanemisega sortimise näide.

Selle sorteerimise korral peab "viit" läbima massiivi pikkusega võrdse pikkus. Samuti võib iga element läbida pikkuse, mis võib ulatuda massiivi enda pikkuseni. Seega võib halvimal juhul algoritmi ajaline keerukus ulatuda $O(n^2)$. Keskmiselt on sortimisalgoritmil ka $O(n^2)$. Parimal juhul on algoritmil $O(n)$ ajaline keerukus.

Selle algoritmi rakendamisel ei kaasata täiendavaid parameetreid ja ainus toimuv toiming on muutujate kohtade asendamine massiivis, seetõttu on sellisel algoritmil ruumi keerukus $O(1)$.

Tulenevalt asjaolust, et algoritm läbib massiivi sammhaaval ja asendab elemente ainult siis, kui üks elementidest on suurem kui teine, siis on selline algoritm stabiilne.

```
void insertionSort(int* arr, size_t length) {
	for (int i = 1, j = 0, tempNum; i < length; i++) {
		tempNum = arr[i];
		for (j = i - 1; j >= 0 && arr[j] > tempNum; j--)
arr[j + 1] =arr[j];
		arr[j + 1] = tempNum;
	}
}
```

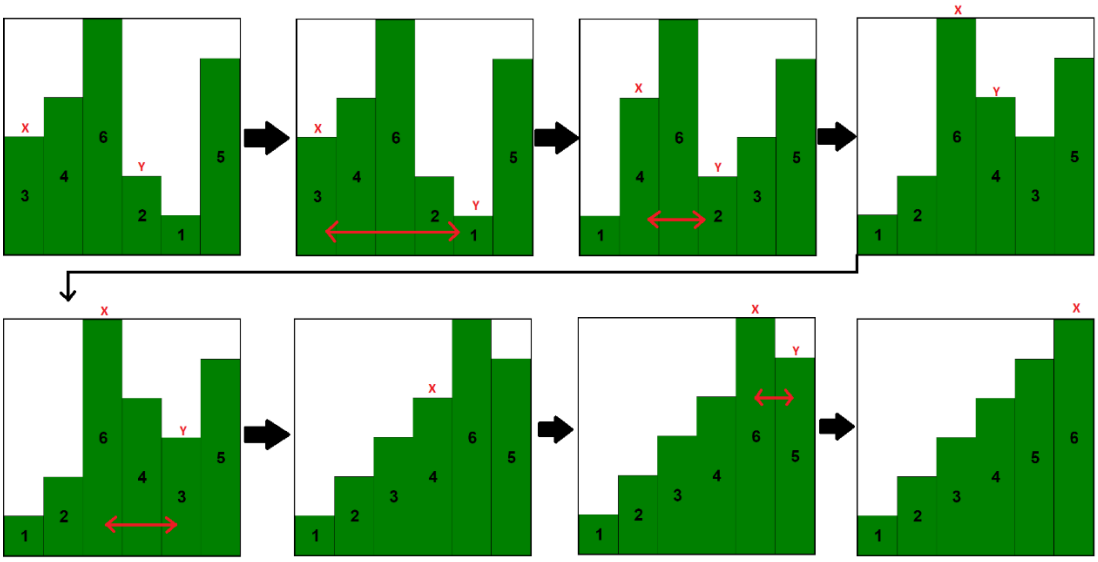
Joonis 9. Vahelepanemisega sortimise algoritmi rakendamine

Vahelepanemisega sorteerimisalgoritm on väga hõlpsasti mõistetav ning andmete sorteerimise kood ise on äärmiselt lihtne ja võtab vähem kui 10 rida. (vt Joonis 9)

1.3.3 Valiksortimine

Valiksortimine on veel üks mullsortimise ja vahelepanemisega sortimise võrdväärne elementaarne sorteerimise algoritm.

Kui mullsortimise abil täitis algoritm massiivi samm-sammult lõpust, see tähendab, et alguses otsis see väärtuse järgi suurimat elementi ja pani selle massiivi lõppu, siis valiksortimise korral algab algoritm kõige esimesest elemendist massiivis. “Viit” algab massiivi esimesest elemendist ja liigub samm-sammult viimase elemendi juurde. Sel juhul, alates esimesest elemendist, kontrollib algoritm ka kõiki järgnevaid elemente ja kui algse elemendi (olgu see nimeks X) väärtus on suurem kui järgmise kontrollitava elemendi (olgu selle nimeks Y), siis algoritm säilitab seda kohta ja pärast seda hakkab algoritm järgnevaid elemente võrdlema elemendiga Y ja kui Y on järgneva elemendi väärtus suurem, siis nüüd saab sellest järgnevast elemendist element Y, mida algoritm säilitab. Kui algoritm jõuab viimase elemendini, siis kui algoritm leiab sellise elemendi Y, vahetab see X elemendi ja Y elemendi. Seejärel liigub algoritm järgmise elemendi juurde [6] [7].



Joonis 10. Valiksortimise näide

Kuna selle sortimise korral peab algoritm minema massiivi algusest lõpuni n korda, kus n on massiivi pikkus, ning samuti sorteerima iga üksiku numbri, millest massiivis on n tükki (vt Joonis 10), siis see algoritm, sama mis mullsortimiini puhul, on aja keerukus $O(n^2)$ [7].

Selle algoritmi rakendamisel ei kaasata täiendavaid parameetreid, mis sõltuksid massiivi suurusest, seetõttu on algoritmi ruumi keerukus O(1).

Pealegi võtavad parameetrid ajas keerukus ja ruumi keerukus parimal, keskmisel ja halvimal juhul vastavalt $O(n^2)$ ja O (1) koha, sest parimal juhul ja halvimal juhul peab algoritm läbima sama palju elemente.

Tulenevalt asjaolust, et algoritm asendab numbreid ainult siis, kui esimene number on suurem kui teine, on selline algoritm stabiilne.

Valiksortimine sorteerimisalgoritm on väga hõlpsasti mõistetav ja ka rakenduskood ei võta palju ruumi. (vt Joonis 11)

```
void selectionSort(int* arr, size_t length) {
	int tempIndex, tempNum;
	for (int i = 0; i < length - 1; i++) {
		tempIndex = i;
		for (int j = i + 1; j < length; j++) {
			if (arr[j] < arr[tempIndex])
				tempIndex = j;
		}
		tempNum = arr[i];
		arr[i] = arr[tempIndex];
		arr[tempIndex] = tempNum;
	}
}
```

Joonis 11. Valiksortimise algoritmi rakendamine

Parima juhul kindlakstegemiseks ja selle aja keerukuse võrdseks O(n)-ga saab seda algoritmi, nagu mullsortimine, parandada. (vt Joonis 12)

```
void selectionSort(int* arr, size_t length) {
	int tempIndex, tempNum, swap = 0;
	for (int i = 0; i < length - 1; i++) {
		tempIndex = i;
		for (int j = i + 1; j < length; j++) {
			if (arr[j] < arr[tempIndex]) {
				tempIndex = j;
				swap = 1;
			}
		}
		if (!swap) break;
		tempNum = arr[i];
		arr[i] = arr[tempIndex];
		arr[tempIndex] = tempNum;
	}
}
```

Joonis 12. Optimeeritud valiksotimine algoritm

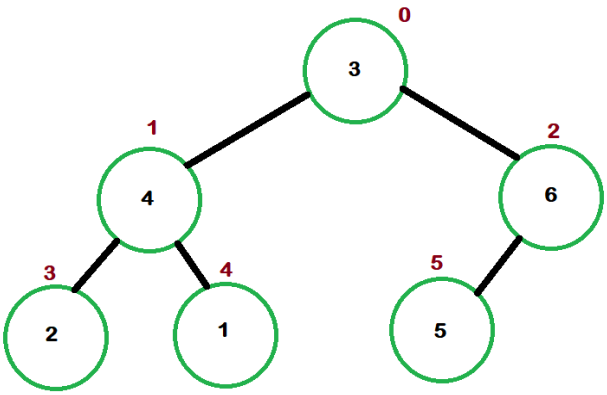
Edasised optimeerimised pole asjakohased.

1.3.4 Kuhja sortimine

Kuhja sortimine on binaarkuhjal põhinev sorteerimisalgoritm. Binaarkuhi on omakorda andmestruktuur, mis on omaduste poolest identne binaarse otsingu puuga, kuid erineb ehituselt. Binaarkuhi võib koosneda mis tahes andmekogumist (massiivist).

Binaarkuhi koostamisel järgitakse kahendotsingupuu reeglit, et puujuure (juure - root) vasak alampuu (vasak sõlm - left node) ja parem alampuu (parem sõlm - right node) võivad olla ka vanempuud (vanemad - parents), mis võivad olla vasakule sõlm ja parempoolne sõlm. Kuid binaarkuhi peab kuhja esimene element olema kas suurim arv ja kõik selle järgnevad alampuud peavad järgima sama reeglit ülejäänud elementidega (Max heap) või esimene element peab olema väikseim arv ja vastavalt peavad seda reeglit järgima ka kõik selle alampuud (Min heap) [8].

Selle sortimisalgoritmi jaoks kasutatakse binaarset kuhja Max heap. Mistahes massiivi binaarkuja kujutamiseks, mille pikkus on n elementi, on vaja selle jagada alampuudeks. Sel juhul leitakse viimase vanema alampuu indeks (massiivi elemendi järjestus) valemiga: indeks = n / 2 - 1 (Loomulikult on kõik elemendid kuni saadud indeksini ka vanemapuud); vanema koht leitakse valemiga: (indeks-1) / 2, kus indeks on binaarkuja praeguse elemendi indeks; puu vasaku sõlme koht massiivis leitakse valemiga: (2 * indeks) + 1; Massiivi puu parema sõlme leiatakse valemiga: (2 * indeks) + 2 [8] [9] [10].



Joonis 13. Binaarkuja esitus tavalisest massiivist

Selle tulemusena saab massiivi [3, 4, 6, 2, 1, 5] kujutada ülaltoodud binaarse puuga. (vt Joonis 13)

Kõigepealt tuleb see puu korraldada Max heap binaarkuhjaks. Selleks on vaja kontrollida vasak- ja parempoolsete alampuude väärtusi viimasest vanemapuust juureni, nii et vanempuu väärtus oleks suurem kui vasak- ja parempoolne alampuu.

```
void heapify(int *arr, size_t length, int i)
{
	int root = i;
	int tempNum;
	int lChild = 2 * i + 1;
	int rChild = 2 * i + 2;

	do {
		i = root;
		if (lChild < length && arr[lChild] > arr[root])
			root = lChild;

		if (rChild < length && arr[rChild] > arr[root])
			root = rChild;

		if (root != i) {
			tempNum = arr[i];
			arr[i] = arr[root];
			arr[root] = tempNum;
			lChild = 2 * root + 1;
			rChild = 2 * root + 2;
		}
	} while (root != i);
}
```

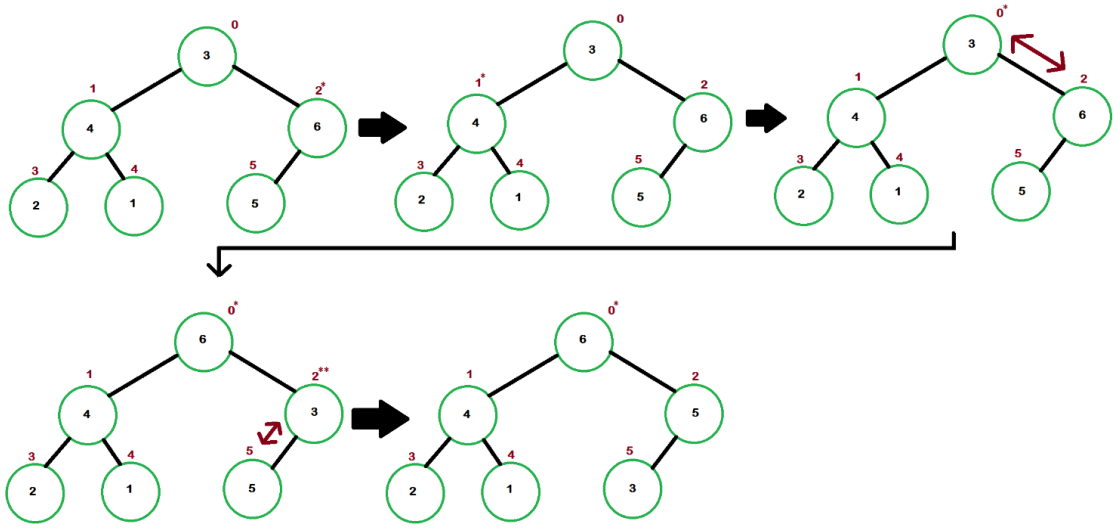
Joonis 14. Heapify alamfunktsiooni rakendamine kuhjasortimise algoritmiks [9]

Sellisel juhul käivitab heapify funktsioon (vt Joonis 14) ülaltoodud algoritmi elemendist “i” kuni viimase võimaliku alampuuni (vt Joonis 15).

```
void heapSort(int* arr, size_t length) {
	int tempNum;
	for (int i = length / 2 - 1; i >= 0; i--)
		heapify(arr, length, i);
	for (int i = length - 1; i > 0; i--) {
		tempNum = arr[0];
		arr[0] = arr[i];
		arr[i] = tempNum;
		heapify(arr, i, 0);
	}
}
```

Joonis 15. Kuhjasortimise algoritmi rakendamine [9]

Binaarse kuhja max heap moodustamise protsess võimaldab täita kuhja sorteerimise algoritmi.



Joonis 16. Kuhja sortortimise näide

Pärast seda, kui algne massiiv on korraldatud max heap binaarkujasse, asendatakse massiivi esimene element viimasega selle tõttu, et binaarhunniku kõige esimene element on massiivi suurim arv. Seega jõuab suurim arv massiivi lõppu, mis tähendab, et antud element on sorteeritud. Pärast seda vähendab algoritm sorteerimata massiivi pikkust ühe võrra ja hakkab saadud puud uuesti sorteerima binaarse kuhjaga max heap-iks ja seejärel esimest elementi viimasega muutma. Ja see jätkub seni, kuni sorteerimata massiivi pikkus on 1, mis tähendab, et massiiv on täielikult sorteeritud. (vt Joonis 15)

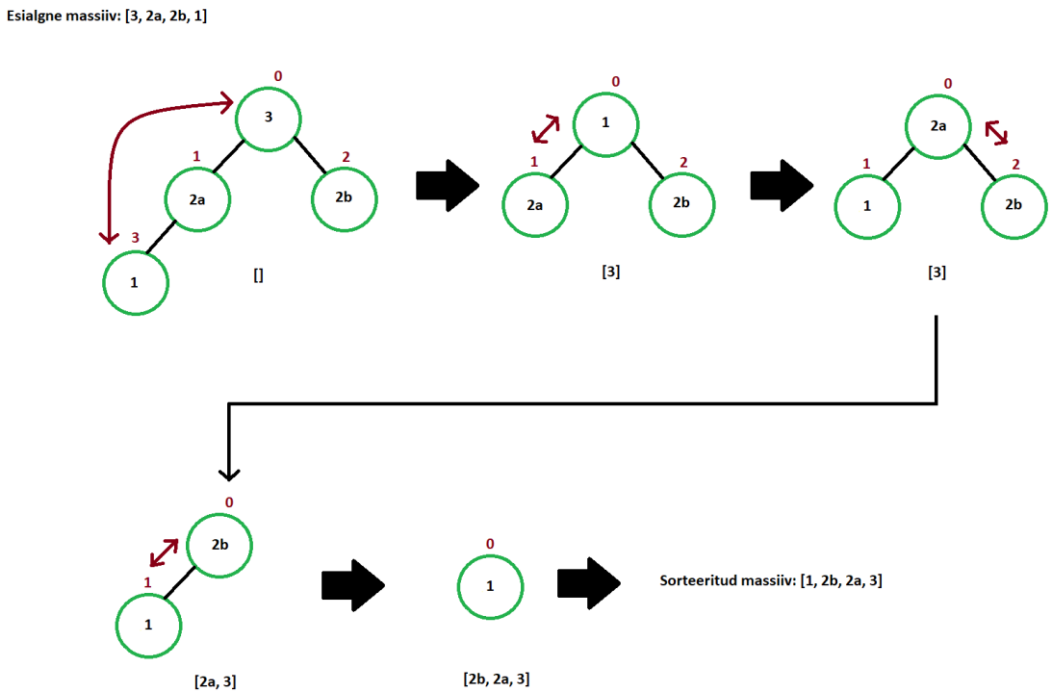
Seega on algoritmil 3 etappi:

1. Korrastada massiiv max heap binaarseks kuhjaks
2. Vahetada massiivi esimene ja viimane element
3. Vähendata massiivi pikkust 1 võrra. Kui massiivi praegune pikkus on suurem kui 1, pöördada tagasi punkti 1 juurde (vt Joonis 15)

Kuna binaarkuhi on binaarne puu, arvutatakse sellise puu kõrgus valemiga $\log_2(n)$, kus n on massiivi pikkus. Sellisel juhul võib max heap binaarkuhja korraldamine võtta sama palju aega kui O(log(n)), kui on vaja elemendid puu juurest muuta puu viimaseks elemendiks. Kuhja sortimise algoritm ise vähendab ka massiivi elementide arvu samm-sammult, mis võtab aega O(n). Kokku võtab kuhja sorteerimise algoritm O(n*log (n)) aja keerukuse. Kuna see algoritm korraldab kuhja ise, võtab algoritm isegi siis, kui algne massiiv oli sorteeritud, O (n*log (n)) selle sortimiseks aega. Seega, hoolimata sellest, kas massiiv on juba sorteeritud (st parim juhtum), keskmine või halb juhtum, võtab algoritm sortimiseks siiski O (n*log (n)) aega. Kuid tuleks selgitada, et kuhja sorteerimise algoritmi puhul on mõni parim juhtum, mis võimaldab saavutada O(n) aja keerukuse. See on juhtum, kui kõik elemendid on sellises järjestuses, et algoritmi ainus toiming on esimese elemendi muutmine viimasega ja sortimata massiivi pikkuse vähendamine 1 võrra [8] - [10].

Ruumi keerukuse seisukohalt ei loo see algoritm massiivi pikkusest sõltuvalt täiendavaid elemente, mis nõuavad lisamälu, mis tähendab, et algoritmi ruumi keerukus on O(1).

Kuhjasortimise algoritm on ebastabiilne. (vt Joonis 17)



Joonis 17. Tõend kuhja sortimisalgoritmi ebastabiilsuse kohta

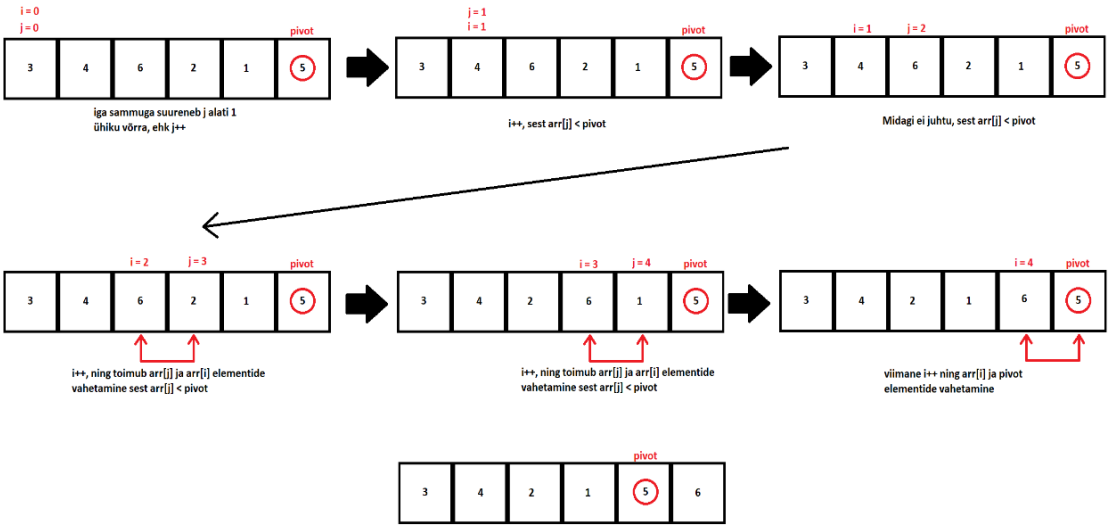
Kuhjasortimise algoritm on keerulisem kui mullsortimine jms ja seetõttu on selle algoritmi kood palju suurem kui sama mullsortimiini kood.

1.3.5 Kiirsortimine

Kiirsortimine on rekursiivne sortimisalgoritm, mis sorteerib elemendid, jagades massiivi pidevalt väiksemateks tükkideks.

Kiirsortimiini kogu mõte on see, et võetakse mõni arv (pöördetapp - pivot) ja kõik arvud, mis on sellest numbrist väiksemad, sorteeritakse massiivi vasakule poole ning kõik sellest numbrist suuremad numbrid sorteeritakse sellest numbrist massiivi paremal pool. Pärast seda asetatakse pöördetapp keskele ja varem kirjeldatud algoritmi korratakse massiivi selle osa jaoks, mis on pöördetapist vasakul, ja massiivi selle osa jaoks, mis on pöördetehasest paremal [11] [12]. (vt Joonis 18)

Kiirsortimine on palju erinevaid variatsioone. Kõigi valikute erinevus seisneb selles, kuidas pöördetapp valitakse. Käesolevas töös käsitlatakse kiirsortimise variatsiooni, kus pöördetappiks on valitud massiivi viimane element.



Joonis 18. Andmete sortimine pöördetappist vasakule ja paremale

Kiirsortimiini rakendamiseks kasutatakse partition abistaja funktsiooni (vt Joonis 19), mille olemus on numbrite jagamine 2 osaks.

```
int partition(int* arr, int start, int end) {
	int pivot = arr[end];
	int tempNum;
	int i = (start - 1);

	for (int j = start; j <= end - 1; j++)
	{
		if (arr[j] < pivot)
		{
			i++;
			tempNum = arr[i];
			arr[i] = arr[j];
			arr[j] = tempNum;
		}
	}
	i++;
	tempNum = arr[i];
	arr[i] = arr[end];
	arr[end] = tempNum;
	return i;
}
```

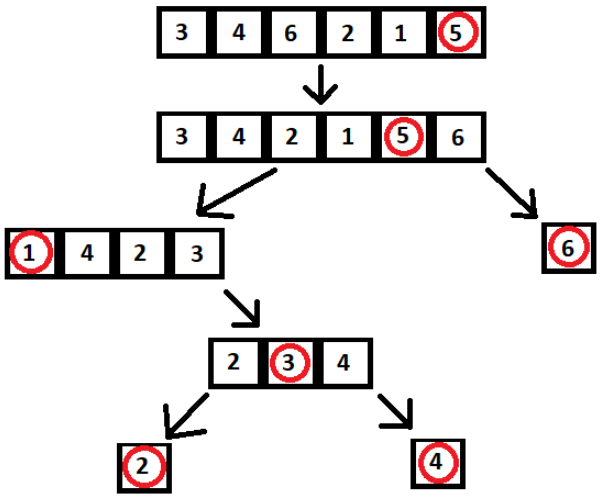
Joonis 19. Partition alamfunktsiooni rakendamine kiirsortimise algoritmiks [13]

Kiirsortimist rakendava funktsiooni quickSort kutsumine toimub rekursiivselt mitu korda. (vt Joonis 20)

```
void quickSort(int* arr, int start, int end) {
	if (start >= end) return;
	int divider = partition(arr, start, end);
	quickSort(arr, start, divider - 1);
	quickSort(arr, divider + 1, end);
}
```

Joonis 20. Kiirsortimise algoritmi rakendamine [13]

Seega, kasutades sama funktsiooni rekursiivset kutsumist ja jagades numbreid kaheks osaks, sorteeritakse kõik andmed õiges järjekorras. (vt Joonis 21)



Joonis 21. Kiirsortimise näide

Selle algoritmi puhul on parim juhtum siis, kui pärast jagava elemendi määramist jagab algoritm massiivi kaheks väiksemaks, võrdse pikkusega massiiviks. Sellisel juhul võib seda stsenaariumi mõelda kui kahendpuud, mille kõrguseks saab $\log_2(n)$. Seega, arvestades algset massiivi, on algoritmi ajaline keerukus parimal juhul O(n*log (n)). Halvimal juhul eemaldab massiiv massiivi jagamise asemel tsüklist ühe massiivi elemendi, seeläbi on algoritmi keerukus $O(n^2)$. Keskmiselt võtab algoritm sortimiseks O(n*log (n)) aega [11].

Kuna tööprotsessi käigus kutsub algoritm ennast rekursiooni abil, siis kulub iga sellise funktsiooni töö säilitamiseks teatud arv arvutimälu. Kuna parimal ja keskmisel juhul on algoritm võimeline massiivi jagama 2 enam-vähem võrdseks massiiviks, on sellise algoritmi ruumiline keerukus O(log(n)). Halvimal juhul, kui algoritm ei jaga massiivi kaheks võrdseks osaks, vaid eemaldab korraga ühe elemendi, luuakse rekursioon, mille pikkus on n, kus n on massiivi pikkus. Sel juhul on algoritmi ruumi keerukus O(n) [13].

Kiirsotimine algoritm on ebastabiilne [13].

See algoritm on pisut keerukama ülesehitusega kui mullsortimine, kuna see algoritm kasutab rekursiooni. Sellise algoritmi kood osutub aga mahukaks.

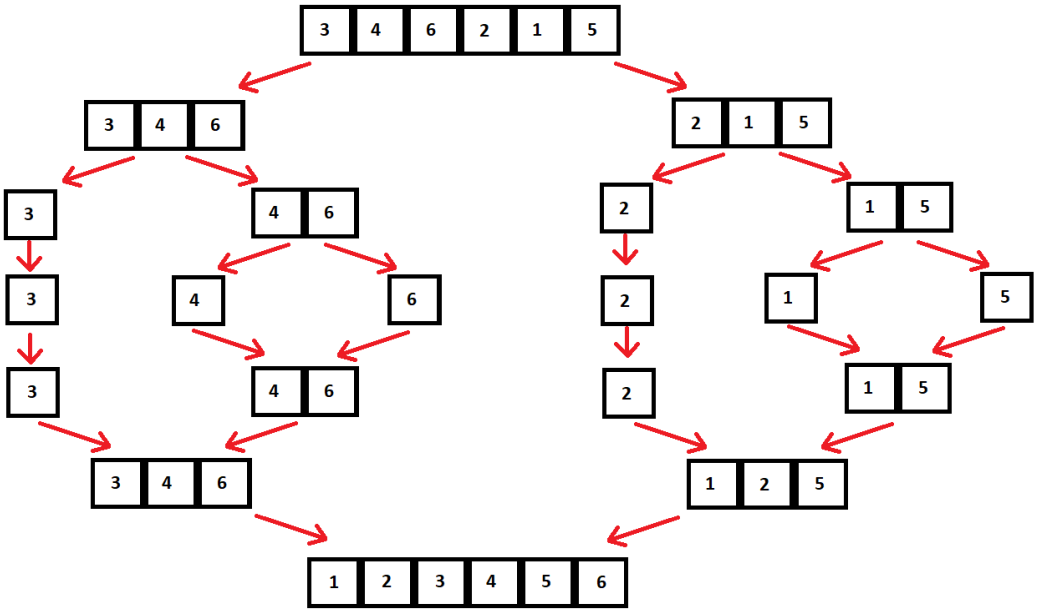
1.3.6 Mestimissortimine

Mestimissortimine on sorteerimisalgoritm, mis kasutab massiivi järjestamiseks nn üheldamist [14].

Sortimisalgoritm mestimissortimine jaguneb 3 osaks:

1. Jagada massiiv võrdseteks osadeks (kui võimalik), kuni algoritm sõelub massiivi eraldi elementideks.
2. Teha elementide üheldamine. See tähendab, et ühendata üksikud elemendid elementide paarideks (ja ühendata elementide paarid suuremaks elementide loendiks), ühendades samal ajal üksikud elemendid õiges järjestuses (see tähendab väikseimast suurimani).

Korrata punkti 2, kuni kaks viimast elementide loendit on ühendatud, sorteerides massiivi. (vt Joonis 22)



Joonis 22. Mestimissortimise näide

Programmeerimise seisukohalt on mestimissortimine üles ehitatud algse funktsiooni rekursiivsele kordamisele, kuni massiiv on jagatud eraldi numbriteks [14]. (vt Joonis 23)

```
void mergeSort(int* arr, int iLeft, int iRight) {
	if (iLeft < iRight) {
		int iMiddle = (iLeft + iRight) / 2;
		mergeSort(arr, iLeft, iMiddle);
		mergeSort(arr, iMiddle + 1, iRight);
		merge(arr, iLeft, iMiddle, iRight);
	}
}
```

Joonis 23. Mestimissortimise algoritmi rakendamine [15]

Pärast massiivi jagamise funktsiooni algab liitmise teine etapp, mida nimetatakse merge. (vt Joonis 24)

```
void merge(int *arr, int iLeft, int iMiddle, int iRight)
{
	int n1 = iMiddle - iLeft + 1;
	int n2 = iRight - iMiddle;

	int *LeftArr = (int*)calloc(n1, sizeof(int));
	int *RightArr = (int*)calloc(n2, sizeof(int));

	for (int i = 0; i < n1; i++) LeftArr[i] = arr[iLeft + i];
	for (int j = 0; j < n2; j++) RightArr[j] = arr[iMiddle + 1 + j];

	int i = 0, j = 0, k = iLeft;
	for (; i < n1 && j < n2; k++) {
		if (LeftArr[i] <= RightArr[j]) {
			arr[k] = LeftArr[i];
			i++;
		}
		else {
			arr[k] = RightArr[j];
			j++;
		}
	}

	for (; i < n1; i++, k++) arr[k] = LeftArr[i];

	for (; j < n2; j++, k++) arr[k] = RightArr[j];
}
```

Joonis 24. Merge alamfunktsiooni rakendamine mestimissortimise algoritmiks [15]

Merge funktsiooni olemus on see, et see loob iga massiivi jaoks eraldi ajutised massiivid LeftArr ja RightArr. Ja pärast seda hakkab funktsioon neid ajutisi massiive kokku koguma ja seejärel andmeid algsesse massiivi kirjutama.

Massiivi eraldi elementideks sõelumiseks kulub maksimaalselt $\log_2(n)$ aeg, kus n on massiivi pikkus. Pärast seda võtab massiivi elementide sorteerimine n aega, sest kõik elemendid on sorteeritud lineaarselt. Seetõttu on selle algoritmi ajaline keerukus

O(n*log(n)). Veelgi enam, sõltumata elementide asukohast massiivis, teostab algoritm sama protseduuri, nii et selle massiivi jaoks pole selliseid mõisteid nagu parim ja halvim juht [15].

Kuna see algoritm sõelub massiivi eraldi elementideks, eraldatakse iga üksiku elemendi jaoks lisamälu, mis sõltub massiivi suurusest. Seetõttu on sellise algoritmi ruumiline keerukus O(n) [15].

See sortimisalgoritm on ka stabiilne.

Keerukuse poolest sarnaneb algoritm kuhjaga sortimisega, mida tõendab nende kahe sortimisalgoritmi peaaegu identne koodimaht.

1.3.7 Timsortimine

Timsortimine on hübriidne sorteerimisalgoritm, mis kasutab nii vahelepanemisega sortiminse kui ka mestimissortimise meetodit "üheldamine".

Selle algoritmi eripära on see, et algoritm jagab massiivi ajutisteks massiivideks, millest igaühel on 32 elementi või 64 elementi [16, p. 34]. Pärast seda sorteeritakse kõik need ajutised massiivid vahelepanemisega sortimisega (vt jaotis 1.3.2) ja väikesed sorteeritud massiivid hakkavad mestimissortimiiniga "üheldata" ühte sorteeritud massiivi (vt jaotis 1.3.6).

Timsortimiini eripära on see, et väikeste massiivide jaoks kasutatakse vahelepanemisega sortimiini efektiivsust ja suurte massiivide sorteerimiseks mestimissortimiini efektiivsust. (vt Joonis 25)

```
void timSort(int *arr, int n)
{
	int size, iLeft, iMiddle, iRight;
	const int run = 32;
	for (int i = 0; i < n; i += run)
		insertionSort(arr, i, min((i + 31), (n - 1)));
	for (size = run; size < n; size = 2 * size)
	{
		for (iLeft = 0; iLeft < n; iLeft += 2 * size)
		{
			iMiddle = iLeft + size - 1;
			iRight = min((iLeft + 2 * size - 1), (n - 1));

			merge(arr, iLeft, iMiddle, iRight);
		}
	}
}
```

Joonis 25. Timsortimise algoritmi rakendamine

Selle algoritmi ajaline keerukus on O(n*log(n)). Parimal juhul, kui massiiv on juba sorteeritud, võtab keerukuse aeg ainult O(n) [16, p. 42] [17, p. 2].

Tulenevalt asjaolust, et algoritm kasutab mestimissortimiini, võtab algoritmi ruumiline keerukus O(n), ja see algoritm on stabiilne [16, p. 43].

Timsortimine on äärmiselt keeruline algoritm, kuna see kasutab koguni 2 sorteerimisalgoritmi, seega on selle algoritmi kood mullsortimiiniga võrreldes väga suur.

1.3.8 Introsortimine

Introsortimine on hübriidne sorteerimisalgoritm, mis kasutab kuni 3 sorteerimisalgoritmi kombinatsiooni: kiirsortimine, heap sort, vahelepanemisega sortimine [18].

See algoritm, nagu Timsortimine, jagab massiivi väiksemateks massiivideks. Veelgi enam, massiivi jagamise suurus sõltub otseselt selle suurusest.

```
void Introsort(int* arr, size_t length) {
	int depth = 2 * int(log(double(length - 1)));
	introsort_imp(arr, 0, length - 1, depth);
}
```

Joonis 26. Introsortimise algoritmi rakendamine

Muutuja depth määrab sügavuse, st mitu korda massiivi kiirsortimise abil jagatakse (vt Joonis 26). Pärast jaotussügavuse arvutamist algab massiivi sorteerimine.

```
void introsort_imp(int* arr, int start, int end, int depth) {

	if (end - start < 16) {
		insertionSort(arr, start, end);
		return;
	}

	if (depth == 0) {
		_heapSort(arr, start, end);
		return;
	}

	int divider = partition(arr, start, end);
	introsort_imp(arr, start, divider - 1, depth - 1);
	introsort_imp(arr, divider + 1, end, depth - 1);
}
```

Joonis 27. Inrosort_imp alamfunktsiooni rakendamine introsortimise algoritmiks

Algoritm alustab massiivi jagamist ja kiiruse sortimise printsiibi abil selle sees olevate numbrite sorteerimist, iga kord pärast jagamist, vähendades sügavust ühe võrra. Kui massiiv on jagatud vähem kui 16 numbriks, tekib massiivi väikese osa vahelepanemisega sortimine. Samuti, kui sügavus jõuab nulli, hakatakse massiivi osa sorteerima kuhjuga. Seega osutub tasakaalu saavutamiseks kolme sorteerimisalgoritmi vahel. (vt Joonis 27)

Selle algoritmi ajaline keerukus on O (n * log (n)) [18].

Tulenevalt asjaolust, et algoritm kasutab kiirsortimiini, võtab algoritmi ruumiline keerukus O(log(n)) [18].

See algoritm ei ole stabiilne [18].

Introsortimine on äärmiselt keeruline algoritm, kuna see kasutab kuni 3 sorteerimisalgoritmi, seega on selle algoritmi kood väga suur.

1.4 Sorteerimisalgoritmide teoreetiline analüüs

Allpool kogutud kõik loetletud sorteerimisalgoritmid ja nende parameetrid nende edasiseks teoreetiliseks analüüsiks.

Tabel 2. Sortimialgoritmide omadused

	Time complexity			**Space complexity**			**Stabiilsus**
	Halvim	**Keskmine**	**Parim**	**Halvim**	**Keskmine**	**Parim**	
Mullsortimine	$O(n^2)$	$O(n^2)$	O(n)	O(1)	O(1)	O(1)	jah
Vahelepanemisega sortimine	$O(n^2)$	$O(n^2)$	O(n)	O(1)	O(1)	O(1)	jah
Valiksortimine	$O(n^2)$	$O(n^2)$	O(n)	O(1)	O(1)	O(1)	jah
Kuhjasortimine	O(nlog(n))	O(nlog(n))	O(n)	O(1)	O(1)	O(1)	ei
Kiirsortimine	$O(n^2)$	O(nlog(n))	O(nlog(n))	O(n)	O(log(n))	O(log(n))	ei
Mestimissortimine	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(n)	O(n)	O(n)	jah
Timsortimine	O(nlog(n))	O(nlog(n))	O(n)	O(n)	O(n)	O(n)	jah
Introsortimine	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(log(n))	O(log(n))	O(log(n))	ei

Kõiki selles töös analüüsitavaid sorteerimisalgoritme on võimalik tinglikult jagada 3 rühma:

1. Elementaarsed sortimisalgoritmid: Mullsortimine, Vahelepanemisega sortimine ja Valiksortimine.
2. Komplekssed sortimisalgoritmid: Kuhja sortimine, Kiirsortimine ja Mestimissortimine
3. Hübriid sortimisalgoritmid: Timsortimine ja Introsortimine

Igal sortimisalgoritmide rühmal on oma eripärased unikaalsed omadused.

Elementaarne sortimisalgoritmiderühm ei kasuta andmete sorteerimiseks lisamälu. Samuti on võimalik märgata, et elementaarsed sorteerimisalgoritmid vajavad tavaliselt 2 tsüklit, mistõttu nende ajaline keerukus ulatub keskmiselt $O(n^2)$-ni. Sellised sortimisalgoritmid on stabiilsed. Nende algoritmide teine eripära on nende üsna lihtne sortimisalgoritm ja suhteliselt väike programmikood. Sellised sortimisalgoritmid sobivad kõige paremini selliste arvutite jaoks, kus programmi võimaliku ROM-i maht on äärmiselt piiratud. Kui võrrelda selle rühma algoritme, siis, nagu mainiti jaotises 1.3.3, on valiksortimine mullsortimiini vastupidine versioon. Vahelepanemisega sortimisel on mullsorteerimisest veidi erinev algoritm ja selline sorteerimisalgoritm võtab vähem aega, kui sorteerimise alguses on massiiv juba peaaegu sorteeritud, kuna mullsorteerimise korral on alati vaja läbida 2 tsüklit. Vahelepanemisega sortimise korral on vaja läbida massiivi kahekordse tsükli abil ainult nii mitu korda, kui massiivis on sorteerimata numbreid. Seetõttu on hübriidsorteerimise algoritmides eelistatud vahelepanemisega sortimine. Kuna tulevastes hübriidalgoritmides kasutatakse vahelepanemisega sortimist, võetakse arvesse ainult sellest grupist vahelepanemisega sortimine, kuna muud sorteerimisalgoritmid on palju aeglasemad kui vahelepanemisega sortimine.

Komplekssede sorteerimisalgoritmide rühmal on keerukamad sorteerimisalgoritmid ja sellised algoritmid püüavad vältida koodis kahekordseid tsüklid, mille tõttu on aja keerukus $O(n^2)$. Seetõttu võivad sellised sorteerimisalgoritmid olla ebastabiilsed ja sortimiseks vajavad ka lisamälu. Kui võrrelda kompleksseid sorteerimisalgoritme omavahel, siis on nähtud, et kuhjasortimine on üsna kiire sorteerimisalgoritm, mis ei kasuta arvutamisel lisamälu, kuid see sorteerimisalgoritm on ebastabiilne. mestimissortimine on ka väga kiire sorteerimise algoritm, kuid erinevalt kuhjasortimisest vajab arvutamiseks palju lisamälu, kuid see sorteerimisalgoritm on stabiilne. Selle rühma kõige ebaefektiivsem on kiirsortimine, mis halvimal juhul võib vajada $O(n^2)$ sorteerimist ning samas on selline algoritm ebastabiilne ja võtab sortimisel natuke lisamälu (Tabel 2).

See on tingitud asjaolust, et kiirsortimine põhineb sama funktsiooni rekursiivsel kutsumisel ja puudub kontroll selle üle, mitu korda tuleks funktsiooni rekursiivselt kutsuda. Sellest rühmast tuleks kontrollida kõiki võimalikke sorteerimisalgoritme, et jälgida nende efektiivsuse muutumist, kui neid algoritme kasutatakse hübriidsorteerimise algoritmides. Teoreetiliste andmete põhjal on see sortimisalgoritmide rühm praktilises arvutuses palju kiirem kui elementaarsorteerimise algoritmide rühm ja on võimalik, et mõned sortimisalgoritmid on sama kiirusega kui hübriidsorteerimise algoritmid.

Hübriidsorteerimise algoritmide rühm on kõige keerukam, kuna sellised algoritmid kasutavad ära teisi sortimisalgoritme, püüdes kohaneda sisendandmete massiivi suurusega. Kui võrrelda hübriidsorteerimise algoritme üksteisega, on nähtud, et timsortimine on palju efektiivsem kui introsortimine, kuna timsortimine on stabiilne ja parimal juhul on selle aja keerukus O(n) (Tabel 2). Introsortimine nõuab sortimisel aga vähem lisamälu kui timsortimine. Hübriidalgoritmid vajavad nende tõhususe hindamiseks täiendavat praktilist analüüsi. Teoreetiliste andmete põhjal on see sortimisalgoritmide rühm praktilistes arvutustes üks kiiremaid, ehkki on võimalik, et Introsortimine võib olla aeglasem kui mõned komplekssede sorteerimisalgoritmide grupi algoritmid.

2 Mikrokontrollerid

Selles peatükis räägitakse mikrokontrolleritest, sellest, kuidas erinevad ettevõtted loovad oma ainulaadsed mikrokontrollerid, samuti viiakse läbi erinevate mikrokontrollerite lühianalüüs, et valida selle töö jaoks mikrokontroller, mis vastaks sissejuhatuses seatud alamülesannetele.

2.1 Mikrokontrolleri olemus

Mikrokontroller on suhteliselt väike integreeritud elektriskeem, mis on programmeeritav arvuti, millel on oma protsessor, arvuti mälu ja mõni sisend- / väljundsüsteem.

Mikrokontrolleri peamine omadus on erinevalt tavalistest personaalarvutitest selle väike suurus ja madal hind. Madalate kulude ja suuruse tõttu on mikrokontrolleri arvuti osade omadused palju halvemad kui tavalise personaalarvuti omadustel.

Mikrokontrolleril on elektroonikas palju erinevaid rakendusi ja näiteks saab seda kasutada seadmena, mis suudaks andurilt andmeid vastu võtta, teha mõningaid arvutusi ja edastada seejärel signaali või andmeid kaugemale ahelast. Tänu mikrokontrolleri väikesele suurusele on sobib see ideaalselt väikeste arvutiseadmete, näiteks nutikellade jaoks.

2.2 Sisseehitatud disain

Täieliku funktsionaalsuse tagamiseks vajab mikrokontroller minimaalset funktsioonide komplekti, et see saaks täita erinevaid käske ja edastada andmet.

Seega on mikrokontrolleri põhikomponendid protsessor, mälu, sisend- / väljundportid, taimerid, katkestused ja mõnikord lisatakse ka ADC ja DAC muundurid. Kõik need osad on integreeritud ühte kiipi, mida nimetatakse mikrokontrolleriks.

Iga mikrokontrolleri põhiosa on selle mikroprotsessor, mida nimetatakse ka CPU-ks. Mikroprotsessor vastutab vastuvõetud käskude dekodeerimise ja vastuvõetud käsu

edasise täitmise eest. Mikroprotsessori tööpõhimõte ei erine kuidagi arvutiprotsessorist. Kuid suurim erinevus mikrokontrolleri protsessori ja arvuti tavalise protsessori vahel on selle madal jõudlus ja loomulikult väike suurus. Kui arvuti tüüpilise protsessori suurus on ligikaudu 45 mm x 52 mm (intel-i9-7900x) [19], siis on mikrokontrolleri enda suurus 36 mm x 3,8 mm (ATmega328P). Samuti, kui tavaliste arvutite uute protsessorite kiirus arvutatakse GHz-des ja uute protsessorite taktsagedus on üle 3 GHz, siis mikrokontrolleri protsessori taktsagedus varieerub mudeliti, kuid on umbes 1–100 MHz.

Mis tahes mikrokontrolleri tööks on vaja mälu, kuhu mikrokontrolleri protsessor andmeid salvestab. Esiteks kasutatakse mikrokontrollerites mõnda ROM-mälu, kuhu mikrokontrolleri tööprogrammi saaks salvestada [20]. Tavaliselt kasutatakse nendel eesmärkidel mikrokontrolleris nn välkmälu. Sellise mälu eripära on see, et tegemist on püsimäluga, mis tähendab, et isegi pärast elektrikatkestust hoitakse andmeid seadmes ja salvestatud andmeid saab taasesitada, kui seade on uuesti voolu andnud [21]. Samuti on igas mikrokontrolleris mõni RAM-mälu, kuhu arvutamise ajal salvestatakse "ajutised muutujad". Loomulikult pole mikrokontrolleril ühtegi kõvaketast, näiteks HDD või SSD.

Teine oluline mikrokontrollerite sisseehitatud funktsioon on nende taimerid ja katkestused. Taimeri põhiülesanne on iga aja arvestamine. Veelgi enam, tavaliselt töötavad taimerid CPU-st erineva taktsagedusega ja neid rakendatakse riistvaratasandil, seega ei sega taimerid mingil viisil tavalise programmi tööd ega häiri seda mingil viisil. Näiteks on igas mikrokontrolleris nn valvuritaimer (watchdog timer), mis aitab süsteemi külmumisel. Valvekoera taimer töötab nii, et ta saab teatud aja möödudes süsteemilt pidevalt signaali, kuid kui see taimer vastust ei saa, taaskäivitab ta süsteemi sunniviisiliselt [20]. Mikrokontroller sisaldab ka tavalisi taimereid, mida saab käsitsi programmeerida. Tavaliselt kasutatakse taimereid koos katkestustega. Katkestus on signaal, mille protsessorile on saatnud kas riistvara või tarkvara, mis vajab viivitamatut tähelepanu. Riistvaralised katkestused hõlmavad kõiki lisatud kiibist pärinevaid elektroonilisi signaale, mis käivitavad katkestused [20]. Tarkvarakatkestused hõlmavad katkestusi, mis kutsuti mikrokontrollerisse laaditud programmist. Kui äkki käivitatakse katkestus, peatab CPU programmi ja salvestab selle peatamise hetke ning hakkab täitma „katkestamise“ ülesannet. Pärast ülesande täitmist naaseb protsessor tavapärasesse töösse ja jätkab täitmata ülesannete täitmist [22].

Mikrokontrolleri viimased olulised osad on sisend- / väljundportid. Tavaliselt rakendatakse mikrokontrolleris neid “jalgide” kontaktide abil. Kuna sisendsignaalid võivad olla püsiva ja muutuva pingega, töötavad seetõttu mõned kontaktid alalisvoola ja osa vahelduvvoola.

Lisaks võivad mõnel mikrokontrolleril olla ADC ja DAC muundurid. ADC muunduri põhiülesanne on teisendada analoogsignaal ehk pinge digitaalseks numbriteks. DAC-muundur teisendab seadme digitaalsignaali analoogsignaaliks.

Mikrokontrollereid saab programmeerida nii Assembleri programmeerimiskeeles kui ka kõrgema taseme programmeerimiskeeltes nagu C või Python.

2.3 Populaarsed mikrokontrollerite perekonnad

Selles alapeatükis kirjeldatakse mõningaid populaarseid mikrokontrollerite perekondi ja mikrokontrollerite jaoks kasutatavaid tehnoloogiaid.

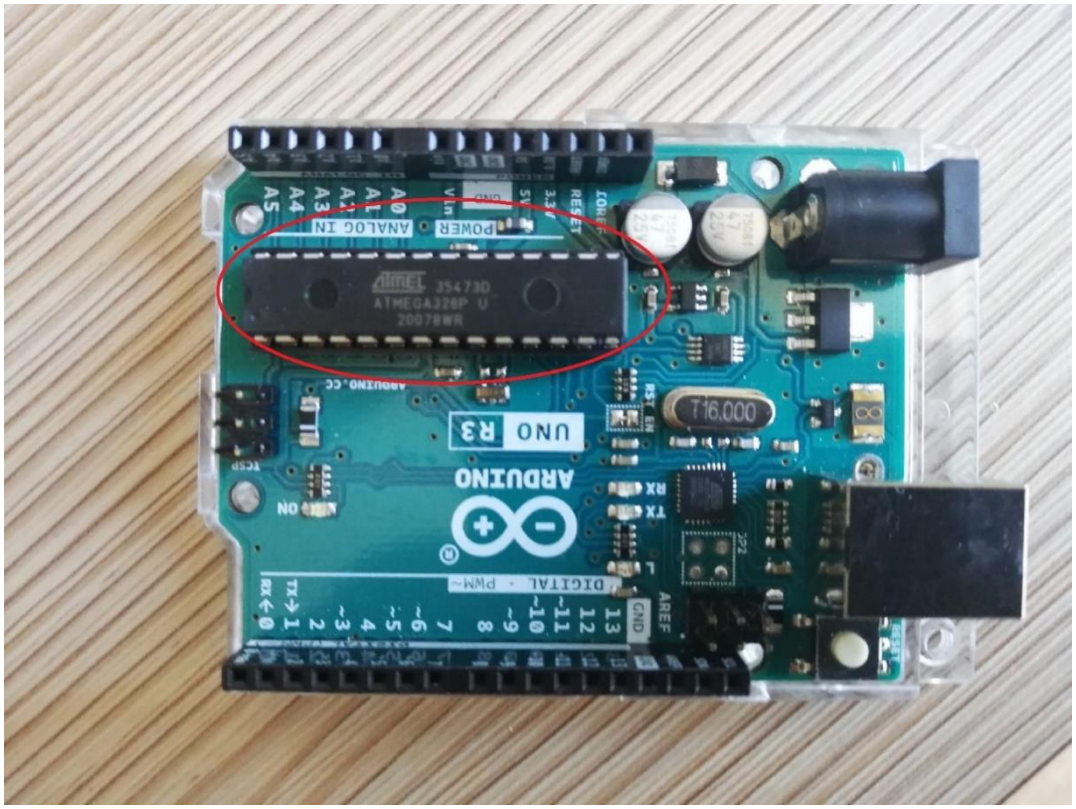
2.3.1 ARM

ARM on protsessorite perekond, mis kasutab RISC-arhitektuuri. ARM toodab 32- ja 64-bitiseid RISC mitmetuumalisi protsessoreid [23]. Tänu oma mikroarhitektuurile on ARM-protsessoritel 32-bitine käsustik, kuid on olemas ka spetsiaalne tihendatud 16-bitine käsustik Thumb [24]. See tähendab, et iga Assembleri käsk saab olema täpselt 32 või 16 bitti. RISC-arhitektuuri tõttu sobivad sellised protsessorid väga hästi mikrokontrollerite tootmiseks, mis kasutaksid ARM-protsessoreid. Nii ilmus ARM Cortex M arhitektuuride seeria, mis sobis mikrokontrollerite tootmiseks ja mida kasutatakse paljudes mikrokontrollerites.

2.3.2 AVR

AVR on populaarne mikrokontrollerite perekond, mis kasutab 8-bitist RISC-arhitektuuri, aga ka 16-bitist ja 32-bitist [24]. AVR-i toodab Atmel. Erinevalt ARM-ist kasutab AVR ainult 16-bitist käskude komplekti. Samuti täidetakse kõik mikrokontrolleri juhised ühe tsüklina. AVR-l on 2 populaarset sarja: ATtiny ja ATmega. ATtiny arhitektuur on väike, kompaktne ja seetõttu suhteliselt väiksem kui ATmega. ATmega arhitektuur pakub rohkem võimalusi ja sisend- / väljundporte, kuid on samal ajal suurem kui ATtiny. Üks põhjus, miks AVR mikrokontrollerid on populaarsust kogunud, on nende kasutamine

Arduino tahvlites (vt Joonis 28), mis on mõeldud mikrokontrollerite mugavamaks kasutamiseks igasuguste projektide jaoks.



Joonis 28. AVR mikrokontroller Arduino trükkplaadil

Arduino pakub ka oma spetsiaalset lahendust AVR mikrokontrollerite programmeerimiseks oma Arduino IDE rakenduse abil, mis lihtsustab töötamist mikrokontrolleri aadressidega.

2.3.3 PIC

PIC on mikrokontrollerite perekond, mis kasutab 8-, 16- ja 32-bitiseid RISC-arhitektuure. Nii nagu AVR, kasutab ka PIC 16-bitist käskude komplekti (siiski on ka teisi 32-bitiseid käske) [24]. Kuid erinevalt AVR-idest pole PIC-mikrokontrolleritel tootjate ja inimeste tuge nii palju ning seetõttu ei ole sellistel mikrokontrolleritel programmeerimine kuidagi lihtsustatud ning tavaliselt kasutatakse selliste mikrokontrollerite jaoks C- ja Assamblee programmeerimiskeeli ning PIC-mikrokontrollereid täitke iga käsk 4 tsüklit.

2.4 Mikrokontrollerite analüüs

Allpool on toodud väike analüüs iga perekonna laialdaselt kasutatavatest mikrokontrolleritest.

Tabel 3. Mikrokontrollerite parameetrid

	ATmega328P [25]	**PIC16F887 [26]**	**ARM Cortex-M4F [27]**
Mikrokontrollerite perekond	AVR	PIC	ARM
ISA	8-bit, RISC	8-bit, RISC	Thumb-2 segatud 16-bit/32-bit arhitektuur, RISC
CPU kiirus	Max 20MHz	Max 20 MHz	Max 80 MHz
Erinevate võimalike unikaalsete käskude arv	131	35	232
Välkmälu (programmi koodi jaoks)	4/8/16/32 kBytes	4/8/16 kBytes	256 kBytes
SRAM mälu	2 kBytes	256 bytes	32 kBytes

Tabelist on nähtud (Tabel 3), et ARM-mikrokontrolleritel on rohkem mälu ja suurem protsessori kiirus, kuid sellised mikrokontrollerid kasutavad vastupidiselt 8-bitistele PIC- ja AVR-arhitektuuridele keerukamat 32-bitist Thumb-2 arhitektuuri.

Väärib märkimist, et PIC-l on äärmiselt vähe võimalikke erinevaid käskusied, mistõttu vajab selline mikrokontroller kõigi käskude jaoks kõige vähem mälumahtu.

ARM-i ja AVR-i mikrokontrollerite väga oluline eelis on võime kohe osta valmis trükkplaat koos SMD-komponentide ja mikrokontrolleriga, mistõttu pole vaja elektroonikaahela täiendavat kokkupanekut seadme täielikuks mikrokontrolleri tööks. Näiteks valmistab Texas Instruments valmis trükkplaate ARM-mikrokontrolleriga, Arduino aga valmis trükkplaate AVR-mikrokontrolleriga. Lisaks võimaldab Arduino Arduino IDE programmi abil mikrokontrolleri kohe tööolekusse seadistada.

ARM-mikrokontrolleril on üsna muljetavaldavad omadused, mistõttu on sellises mikrokontrolleris koormuse loomine sorteerimisalgoritmide abil väga keeruline. Kuid PIC- ja AVR-perekondade mikrokontrollerid võivad sellele tööle sobida.

Tänu AVR mikrokontrolleri omadustele, mis vastatavad püstitud almülesandele, ja ka selle kasutamise mugavusele kasutatakse selles töös Arduino plaadile paigaldatud AVR mikrokontrollerit - ATmega328P.

3 Rakendus

Selles peatükis teostatakse teoreetilise analüüsi käigus 6 esimesest peatükist valitud kiirema sorteerimise algoritmi praktiline analüüs. (vt jaotis 1. Sortimisalgoritmid)

3.1 Ettevalmistus sorteerimisalgoritmide analüüsimiseks

3.1.1 Riistvara ja tarkvara

Nagu on kirjeldatud jaotises 2.4, valiti Arduino sorteerimisalgoritmide analüüsimiseks, millel asub AVR mikrokontroller. Arduino ühendub arvutiga USB-juhtme kaudu ja saadab kõik vajalikud andmed arvuti konsooli. Selle analüüsi jaoks kasutatakse spetsiaalset katkestusfunktsiooniga Arduino taimerit, mille seadistust kirjeldatakse punktis 3.1.2. Ka Arduinos kasutati protsessori toetatud “short” andmetüübi elementide massiivi. “Short” andmetüüp on täisarv. Sellise andmetüübi suurus on 2 baiti ja selline element saab numbreid salvestada vahemikus [−32 767, +32 767].

3.1.2 Programmi kood

Programmi täielikku koodi on võimalik vaadata lisast (vt Lisa 2 - 8)

Sorteerimisalgoritmide analüüsimiseks on loodud abifunktsioonid, mis vastutavad sorteerimisalgoritmi otsese analüüsi eest.

Kõigepealt loodi pseudojuhuslike arvude generaator, et genereerida juhusjärjestuses erinevate numbritega massiiv.

```
void createRandomArr(short arr[], size_t len) {
  srand(analogRead(rand() % 6));
  for (int i = 0; i < len; i++)
    arr[i] = rand() % len;
}
```

Joonis 29. CreateRandomArr funktsiooni rakendamine

Samuti kasutatakse mikrokontrolleris vaba RAM-mälu arvutamiseks freeRAM-i, mis kasutab selle arvutamise jaoks Arduino.h teegi abil deklareeritud muutujaid.

```
int freeRAM() {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

Joonis 30. freeRam funktsiooni rakendamine

Funktsiooni timer_setup (vt Joonis 31) abil luuakse taimer, mis käivitab katkestused sagedusega 2kHz ja katkestuse korral kutsutakse spetsiaalne funktsioon ISR, mis arvutab välja hetkel väikseima vaba RAM-i koguse.

```
void timer_setup() {
  noInterrupts();
//set timer1 interrupt at 2kHz
  TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0
  // set compare match register for 2kHz increments
  OCR1A = 8000;// = (16*10^6) / (2000*1) - 1 (must be <65536)
  // turn on CTC mode
  TCCR1B |= (1 << WGM12);
  // Set CS10 bit for 1 prescaler
  TCCR1B |= (1 << CS10);
  // enable timer compare interrupt
  TIMSK1 |= (1 << OCIE1A);
  interrupts();
}
ISR(TIMER1_COMPA_vect){//timer1 interrupt 1MHz
  currFreeMem = freeRAM();
  if (currFreeMem < prevFreeMem) {
    prevFreeMem = currFreeMem;
  }
}
```

Joonis 31. Taimeri ja katkestuse funktsiooni rakendamine.

Kõige olulisem on funktsioon avgTime (vt Joonis 32), mis arvutab sortimisalgoritmi kohese tööaja ja kasutatud mälu hulga. Esimene For-tsükkel kordatakse 100 korda ja iga kord kirjutatakse massiivi sorteerimiseks kuluv aeg “sum” muutujasse. Seejärel jagatakse see muutuja 100-ga ja saadakse massiivi keskmine sortimise aeg. Pärast seda algab taimer ja algab teine For-tsükkel, mis samuti kordatakse 100 korda ja sorteerib massiivi 100 korda. Seega leitakse ISR-funktsiooni kasutades väikseim vaba mälumaht ja arvutatakse mälumaht, mille programm massiivi sorteerimiseks eraldab.

```
void avgTime (short arr[], short len, void (*sortAlg)(short [], short, short))
{
  unsigned long start_t;
  double sum;
  for (short i = 0; i < 100; i++){
    createRandomArr(arr, len);
    start_t = millis();
    (*sortAlg)(arr, 0, len - 1);
    start_t = millis() - start_t;
    sum += start_t;
  }

  timer_setup();

  Serial.println();
  Serial.print("Available RAM before testing: ");
  Serial.print(freeRAM());

  for (short i = 0; i < 100; i++){
    createRandomArr(arr, len);
    (*sortAlg)(arr, 0, len - 1);
  }

  Serial.println();
  Serial.print("Average time to sort array: ");
  Serial.print(sum/100);
  Serial.println();

  Serial.print("Minimum of available RAM: ");
  Serial.print(prevFreeMem);
  Serial.println();
}
```

Joonis 32. Keskmise sorteerimisaja ja vaba RAM mälu arvutamise funktsiooni rakendamine.

Selle tulemusena väljastab funktsioon avgTime pärast kõiki arvutusi andmeid Arduino IDE konsoolile, kust need andmed analüüsimiseks tabelisse kirjutati. (vt Tabel 4)

3.2 Andmete analüüs

Allpool (vt Tabel 4) kuvatakse tabelis kõik andmed, mis saadi iga valitud sorteerimisalgoritmi rakendamisel.

Tabel 4. Analüüsiks saadud andmed

	Aeg massiivi sortimiseks (ms)			**Kasutatud RAM (baitid)**		
Elementide arv massiivis	100	375	750	100	375	750
Vahelepanemisega sortimine	3,14	42,40	169,21	43	43	43
Kuhja sortimine	3,70	18,80	41,11	43	43	43
Kiirsortimine	1,55	7,11	15,82	109	162	188
Mestimissortimine	7,38	31,08	-	292	793	-
Timsortimine	2,27	12,60	-	260	811	-
Introsortimine	1,72	7,96	17,86	107	169	214

Analüüsi jaoks võeti 3 eraldi 100-elementilist massiivi (kuna “short” on 2 baiti, võtab selline massiiv kokku 200 baiti RAM-i mälu), 375 elementi (750 baiti RAM-mälu) ja 750 elementi (1500 baiti RAM-i mälu).

Vastavalt saadud tabelile saate sorteerimisalgoritmide teoreetilises analüüsis jälgida kõiki kirjeldatud sõltuvusi (vt jaotis 1.4). Näiteks võtame tabelist 4 “Analüüsiks saadud andmed” 100 ja 375 elementi sorteerimise aja vahelepanemisega sortimise abil. 100 elementi sorteeriti 3,14 ms jooksul. Pärast seda suurendati elementide arvu 375/100 = 3,75 korda. Seega, teades, et vahelepanemisega sortimise abil on keskmine aja keerukus $O(n^2)$ (vt jaotis 1.3.2), sorteeritakse 375 elementi 3,14 ms * $(3,75)^2 \approx 44,16$ ms jooksul. Tabelis on 42,40 ms, mis peaaegu läheneb teoreetilisele arvutusele.

Samuti on võimalik märgata, et kuhja sortimisel ja vahelepanemisega sortimisel on sama palju kasutatud RAM-i mis tahes arvu elementide jaoks. See on tingitud asjaolust, et andmete ruumiline keerukus on O(1) (vt jaotis 1.3.2 ja jaotis 1.3.4), mis tähendab, et sellised sortimisalgoritmid kasutavad sama palju mälu, mis ei sõltu massiivi elementide arv.

Mestimissortimisel ja timsortimisel on graafikus lüngad massiivi 750 elemendi sorteerimisega, kuna need algoritmid nõudsid mikrokontrollerilt nii palju RAM-i, et see peatas töö ja oli vaja süsteemi taaskäivitada.

Kõigi sortimisalgoritmide efektiivsuse võrdlemiseks koostas töö autor 2 valemit, mis võiksid võimalikult täpselt kajastada sorteerimisalgoritmide efektiivsust.

Esimene valem (1) ühendab kasutatud RAM-i ja sortimisaja ühiseks parameetriks.

$$D_{n_1-n_2} = \frac{elementide\_arv}{aeg\_masiivi\_sortimiseks} n_1 + \frac{masiivi\_baitine\_suurus}{kasutatud\_RAM} n_2 \tag{1}$$

$$- kus\ (n_1 + n_2) = 100\%$$

Esimene valem (1) aitab hinnata sama sorteeritavate elementide arvu algoritmide sortimise efektiivsust. Teisisõnu aitab see valem hinnata, millised nende sortimisalgoritmid on tõhusamad, kui peame sorteerima täpselt 100 või täpselt 200 elementi. Ka valemis on olulised muutujad n_1 ja n_2. Need muutujad näitavad, milline osa sortimisest on olulisem - kas sorteerimise aeg või säilitatava RAM mälu maht. See tähendab, et kui muutuja n_1 oleks võrdne 75% -ga, oleks muutuja n_2 võrdne 25% -ga, mis omakorda tähendab, et $D_{75\%-25\%}$ näitab sorteerimisalgoritmi efektiivsust, tingimusel et sortimisaeg on olulisem kui sortimisel kasutatud mälumaht. Loomulikult, mida suurem on saadud arv, seda tõhusam on sorteerimisalgoritm seadme jaoks, millega testid tehakse.

Tabel 5. Erinevate sortimisalgoritmide efektiivsus kus 1- Vahelepanemisega sortimine; 2 - Kuhja sortimine; 3 – Kiirsortimine; 4 – Mestimissortimine; 5 – Timsortimine; 6 – Introsortimine

	Elementide arv	**1**	**2**	**3**	**4**	**5**	**6**
$D_{75\%-25\%}$	100	25,05	21,43	48,85	10,24	33,23	44,07
	375	10,99	19,32	40,71	9,29	22,55	36,44
	750	12,05	22,40	37,55	0	0	33,25
$D_{50\%-50\%}$	100	18,25	15,84	33,18	7,12	22,41	29,08
	375	13,14	18,69	28,69	6,51	15,34	25,77
	750	19,66	26,56	27,69	0	0	24,50
$D_{25\%-75\%}$	100	11,45	10,25	17,51	3,90	11,59	15,94
	375	15,29	18,07	16,66	3,73	8,13	15,11
	750	27,27	30,72	17,84	0	0	15,76

Tabeli ülaservas (vt Tabel 5) arvutatakse kõige olulisemad efektiivsused, mida kasutatakse tulevikus. Need tulemused näitavad juba selgelt sorteerimisalgoritmide tõhusust. Lihtsamaks võrdlemiseks tuleks siiski koostada veel üks valem, mille abil on juba võimalik nendest efektiivsustest järeldus teha.

$$S_{n1-n2} = \frac{\sum_{m=0}^{m}\left(D_{n_1-n_2}^{m} * \frac{l_m}{l_0}\right)}{k} \tag{2}$$

$$- kus\ D_{n_1-n_2}^{0}\ on\ suurima\ elementide\ arvuga\ sortimisalgoritmi$$

$$efektiivsus; l\ on\ sorteeritavate\ elementide\ arv,$$

$$k\ on\ D\ elementide\ arv$$

Teine valem (2) ühendab kogu võimaliku sorteerimisalgoritmide leitud efektiivsuse, et võrrelda sortimisalgoritmide efektiivsust üksteisega. Samal ajal leiutatakse kõigi elementide summa valem selliselt, et arvutustes on kõige väärtuslikum kõige suurema elementide arvuga efektiivsus. Kui kasutada seda valemit saadud tulemustega (vt Tabel 5), on üldvalemis 3 elementi.

$$S_{n1-n2} = \frac{\left(D^0_{n_1-n_2} + D^1_{n_1-n_2} * \frac{375}{750} + D^2_{n_1-n_2} * \frac{100}{750}\right)}{3} \quad (3)$$

Sorteerimisalgoritmide efektiivsuse summa leidmiseks kasutatakse valemit (3).

Tabel 6. Sorteerimisalgoritmide efektiivsuse summa (750 elementi)

	$S_{75\%-25\%}$	$S_{50\%-50\%}$	$S_{25\%-75\%}$
Vahelepanemisega sortimine	6.96	9.55	12.15
Kuhja sortimine	11.64	12.67	13.71
Kiirsortimine	21.47	15.49	9.50
Mestimissortimine	2.01	1.40	0.79
Timsortimine	5.24	3.55	1.87
Introsortimine	19.11	13.80	8.48

Seega on tabeli (vt Tabel 6) tulemuste põhjal juba võimalik teha järeldusi sorteerimisalgoritmide kohta.

3.3 Sortimisalgoritmide praktilise analüüsi järeldus

Vastaval tabeli (vt Tabel 6) kiirsortimine on peamiselt kõige tõhusam sorteerimisalgoritm, kui sortimisalgoritmi jaoks on olulisem sortimise kiirus. Samuti, kui nii sortimiskiirus kui ka kulutatud mälu hulk on olulised atribuudid, siis on kiirsortimine ka optimaalne sorteerimise algoritm. Kuid kui kasutatud mälu hulk on olulisem kui aeg, mis kulub sortimiseks, siis on andmete sorteerimiseks tõhusam kasutada kuhja sortimist.

Sorteerimisalgoritmid mestimissortimise ja timsortimise said tabelis 6 väikese arvu, kuna valem põhineb 750 sorteerimist vajaval elemendil. Siiski tuleb märkida, et need algoritmid on erinevalt kiiresortimisest ja kuhja sortimisest stabiilsed (vt Tabel 2). Ja kui oleks vaja stabiilset sorteerimisalgoritmi, oleksid need algoritmid muidugi tõhusamad.

Kokkuvõte

Vastavalt püstitatud eesmärgile on tehtud sorteerimisalgoritmide analüüs selleks et vastata küsimusele, milline sorteerimisalgoritme sobib mikrokontrollerile kõige paremini, et leida suurimad ja väiksemad arvandmed andmete loendis. Analüüsi tulemusena selgus, et Arduino Uno Rev 3 trükkplaadil asuva mikrokontrolleri ATmega328P jaoks on kiirsortimine ekstreemsuse leidmiseks tõhusam. Kui aga on vaja salvestada rohkem RAM-mälu, siis parem kiirsorteerimise asemel kasutada kuhja sortimist.

Samuti lahendati töö alguses püstitatud probleem. See töö näitab, kuidas on võimalik mikrokontrolleri jaoks valida tõhusama sorteerimisalgoritmi, analüüsides sorteerimisalgoritme kasutades C ja C++ keeli.

Selle töö põhjal on võimalik tulevikus luua võimsama mikrokontrolleri andmebaasi prototüübi, mis võimaldab seda teha. Arvuta selles andmebaasis selle töö valemeid ja arvutusi kasutades tõhusam sorteerimisalgoritm. Vastavalt saadud arvutustele teha mikrokontrolleris kiire ja tõhus andmebaas mikrokontorlleri põhjal.

Kasutatud kirjandus

[1] . S. O'Dea, „Number of smartphone users worldwide from 2016 to 2023," Statista, 31 Märts 2021. [Võrgumaterjal]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Kasutatud 6 Aprill 2021].

[2] S. Woltmann, „Big O Notation and Time Complexity – Easily Explained," HappyCoders, 28 Mai 2020. [Võrgumaterjal]. Available: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/. [Kasutatud 3 Märts 2021].

[3] GeeksforGeeks, „Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)," GeeksforGeeks, 9 November 2020. [Võrgumaterjal]. Available: https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/. [Kasutatud 5 Märts 2021].

[4] GeeksforGeeks, „What does 'Space Complexity' mean?," GeeksforGeeks, 24 Veebruar 2021. [Võrgumaterjal]. Available: https://www.geeksforgeeks.org/g-fact-86/. [Kasutatud 5 Märts 2021].

[5] T. Stringer, „Sorting Algorithm Stability - What it is and When it Matters," Thomas Stringer, 14 Detsember 2020. [Võrgumaterjal]. Available: https://trstringer.com/sorting-stability/. [Kasutatud 3 Märts 2021].

[6] M. Faalil, „Selection Sort," DEV Community, 25 November 2020. [Võrgumaterjal]. Available: https://dev.to/ucscmozilla/selection-sort-3hf7. [Kasutatud 5 Märts 2021].

[7] S. Heydari, „Selection Sort in Python," Stack Abuse, 14 Veebruar 2020. [Võrgumaterjal]. Available: https://stackabuse.com/selection-sort-in-python/. [Kasutatud 5 Märts 2021].

[8] K. M. Rahman, „Binary Heap Note 1," Medium, 7 Detsember 2018. [Võrgumaterjal]. Available: https://medium.com/@iamcrypticcoder/https-medium-com-iamcrypticcoder-binary-heap-note-1-a40c191ce2df. [Kasutatud 5 Märts 2021].

[9] P. Garg, „Heaps and Priority Queues," HackerEarth, 15 Juuli 2015. [Võrgumaterjal]. Available: https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/. [Kasutatud 5 Märts 2021].

[10] S. Woltmann, „Heapsort – Algorithm, Source Code, Time Complexity," HappyCoders, 19 August 2020. [Võrgumaterjal]. Available: https://www.happycoders.eu/algorithms/heapsort/. [Kasutatud 5 Märts 2021].

[11] L. Vogel, „Quicksort in Java - Tutorial," vogella GmbH, 10 Aprill 2016. [Võrgumaterjal]. Available: https://www.vogella.com/tutorials/JavaAlgorithmsQuicksort/article.html. [Kasutatud 6 Märts 2021].

[12] K. Miyaki, „Basic Algorithms — Quicksort,“ Medium, 11 Veebruar 2020. [Võrgumaterjal]. Available: https://towardsdatascience.com/basic-algorithms-quicksort-b549ea9ef27. [Kasutatud 6 Märts 2021].

[13] S. Woltmann, „Quicksort – Algorithm, Source Code, Time Complexity,“ HappyCoders, 22 Juuli 2020. [Võrgumaterjal]. Available: https://www.happycoders.eu/algorithms/quicksort/. [Kasutatud 8 Aprill 2021].

[14] The Educative Team, „Algorithms 101: How to Use Merge Sort and Quicksort in JavaScript,“ Medium, 4 Detsember 2020. [Võrgumaterjal]. Available: https://betterprogramming.pub/algorithms-101-how-to-use-merge-sort-and-quicksort-in-javascript-6d8908562fe0. [Kasutatud 6 Märts 2021].

[15] D. Landup, „Merge Sort in Java,“ Stack Abuse, 28 Oktoober 2020. [Võrgumaterjal]. Available: https://stackabuse.com/merge-sort-in-java/. [Kasutatud 6 Märts 2021].

[16] M. Hulı´n, „Performance analysis of Sorting Algorithms,“ 2017. [Võrgumaterjal]. Available: https://is.muni.cz/th/gp4gz/bc.pdf. [Kasutatud 7 Märts 2021].

[17] N. Auger, V. Jugé, C. Nicaud ja C. Pivoteau, „On the Worst-Case Complexity of TimSort,“ 2018. [Võrgumaterjal]. Available: https://drops.dagstuhl.de/opus/volltexte/2018/9467/pdf/LIPIcs-ESA-2018-4.pdf. [Kasutatud 7 Märts 2021].

[18] GeeksforGeeks, „Know Your Sorting Algorithm | Set 2 (Introsort- C++’s Sorting Weapon),“ GeeksforGeeks, 15 Märts 2019. [Võrgumaterjal]. Available: https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-introsort-cs-sorting-weapon/. [Kasutatud 2 Aprill 2021].

[19] S. Burke, „Intel i9-7900X Die Size & CPU Size (& Full Res Delid Images),“ GamersNexus, 2 Juuni 2017. [Võrgumaterjal]. Available: https://www.gamersnexus.net/news-pc/2943-intel-i9-7900x-die-size-cpu-size. [Kasutatud 21 Märts 2021].

[20] ElProCus Technologies, „Microcontrollers Types & Their Applications,“ Elprocus, 27 Oktoober 2013. [Võrgumaterjal]. Available: https://www.elprocus.com/microcontrollers-types-and-applications/. [Kasutatud 22 Märts 2021].

[21] M. Gudino, „Introduction to Microcontrollers,“ Arrow Electronics, 26 Veebruar 2018. [Võrgumaterjal]. Available: https://www.arrow.com/en/research-and-events/articles/engineering-basics-what-is-a-microcontroller. [Kasutatud 21 Märts 2021].

[22] J. Valvano ja R. Yerraballi, „Chapter 12: Interrupts,“ 2014. [Võrgumaterjal]. Available: http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm. [Kasutatud 21 Märts 2021].

[23] S. Bigelow, „ARM processor,“ TechTarget, Jaanuar 2015. [Võrgumaterjal]. Available: https://whatis.techtarget.com/definition/ARM-processor. [Kasutatud 22 Märts 2021].

[24] V. Bandlamudi, „Getting Started with Microcontrollers,“ WTWH Media, 27 September 2014. [Võrgumaterjal]. Available: https://www.engineersgarage.com/tutorials/getting-started-with-microcontrollers/. [Kasutatud 22 Märts 2021].

[25] Microchip Technology Inc., „megaAVR® Data Sheet,“ 2018. [Võrgumaterjal]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf. [Kasutatud 4 Aprill 2021].

[26] Microchip Technology Inc., „PIC16F882/883/884/886/887 Data Sheet,“ 2015. [Võrgumaterjal]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/40001291H.pdf. [Kasutatud 4 Aprill 2021].

[27] Texas Instruments Inc., „Tiva™ TM4C123GH6PM Microcontroller Data Sheet,“ 12 Juuni 2014. [Võrgumaterjal]. Available: https://www.ti.com/lit/ds/spms376e/spms376e.pdf?ts=1617519195968&ref_url=https%253A%252F%252Fwww.google.com%252F. [Kasutatud 4 Aprill 2021].

[28] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

[29] M. Silva, „Introduction to Microcontrollers - Timers,“ EmbeddedRelated, 27 September 2013. [Võrgumaterjal]. Available: https://www.embeddedrelated.com/showarticle/478.php. [Kasutatud 21 Märts 2021].

[30] A. Karagodov, „Document,“ -, 10 Märts 2021. [Võrgumaterjal]. Available: https://htmlpreview.github.io/?https://github.com/AlexKar0706/Sorting/blob/main/index.html. [Kasutatud 20 Märts 2021].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Aleksei Karagodov

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Sortimisalgoritmide analüüs ekstremumite leidmiseks mikrokontrolleri rakendustes", mille juhendaja on Vladimir Viies
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

12.05.2021

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 - Programmkood sortimisalgoritmide analüüsimiseks

Programm on jagatud päisefailiks (.h) ja programmifailiks (.cpp).

```
#ifndef arrayFunc_h
#define arrayFunc_h
#include <Arduino.h>
#include <stdlib.h>
#include <time.h>
int freeRAM();
void printArr(short arr[], size_t len);
void createRandomArr(short arr[], size_t len);
void checkArr(short arr[], size_t len);
void avgTime (short arr[], short len, void (*sortAlg)(short [], short, short));
#endif
```

Joonis 2-1 arrayFunc.h faili kood

```
#include "arrayFunc.h"

int freeRAM() {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}


void timer_setup() {
  noInterrupts();
  //set timer1 interrupt at 2kHz
  TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0
  // set compare match register for 1hz increments
  OCR1A = 8000;// = (16*10^6) / (2000*1) - 1 (must be <65536)
  // turn on CTC mode
  TCCR1B |= (1 << WGM12);
  // Set CS10 bits for 1 prescaler
  TCCR1B |= (1 << CS10);
  // enable timer compare interrupt
  TIMSK1 |= (1 << OCIE1A);
  interrupts();
}

int currFreeMem = 2299;
int prevFreeMem = 2299;

void printArr(short arr[], size_t len) {
  Serial.println();
  Serial.print("[ ");
  for (short i = 0; i < len; i++) {
    Serial.print(arr[i]);
    Serial.print(" ");
```

```
    if (!(i%50) && i != 0) Serial.println();
  }
  Serial.print("]");
}

void createRandomArr(short arr[], size_t len) {
  srand(analogRead(rand() % 6));
  for (int i = 0; i < len; i++)
    arr[i] = rand() % len;
}

void checkArr(short arr[], size_t len) {
  short check = 1;
  for (short i = 0; i < len - 1; i++)
    if (arr[i + 1] < arr[i]) check = 0;
  Serial.println();
  if (check) Serial.print("Array is sorted");
  else Serial.print("Array is not sorted");
}

void avgTime (short arr[], short len, void (*sortAlg)(short [], short, short))
{
  unsigned long start_t;
  double sum;
  for (short i = 0; i < 100; i++){
    createRandomArr(arr, len);
    start_t = millis();
    (*sortAlg)(arr, 0, len - 1);
    start_t = millis() - start_t;
    sum += start_t;
  }

  timer_setup();

  Serial.println();
  Serial.print("Available RAM before testing: ");
```

```
  Serial.print(freeRAM());

  for (short i = 0; i < 100; i++){
    createRandomArr(arr, len);
    (*sortAlg)(arr, 0, len - 1);
  }

  Serial.println();
  Serial.print("Average time to sort array: ");
  Serial.print(sum/100);
  Serial.println();

  Serial.print("Minimum of available RAM: ");
  Serial.print(prevFreeMem);
  Serial.println();
}

ISR(TIMER1_COMPA_vect){//timer1 interrupt 2kHz
  currFreeMem = freeRAM();
  if (currFreeMem < prevFreeMem) {
    prevFreeMem = currFreeMem;
  }
}
```

Joonis 2-2 arrayFunc.cpp faili kood

Lisa 3 – Vahelepanemisega soortimise algoritmi kood

```
#ifndef insertionSort_h
#define insertionSort_h
void insertionSort(short arr[], short start, short end_f);
#endif
```

Joonis 3-1 insertionSort.h faili kood

```
#include "insertionSort.h"

void insertionSort(short arr[], short start, short end_f) {
	for (short i = start + 1, j = 0, tempNum; i <= end_f; i++) {
		tempNum = arr[i];
		for (j = i - 1; j >= start && arr[j] > tempNum; j--) arr[j + 1]
= arr[j];
		arr[j + 1] = tempNum;
	}
}
```

Joonis 3-2 insertionSort.cpp faili kood

Lisa 4 – Kuhja soortimise algoritmi kood

```
#ifndef heapSort_h
#define heapSort_h
void heapify(short arr[], short len, short i);
void heapSort(short arr[], short empty, short len);
#endif
```

Joonis 4-1 heapSort.h faili kood

```
#include "heapSort.h"

void heapify(short arr[], short len, short i)
{
	short root = i;
	short tempNum;
	short lChild = 2 * i + 1;
	short rChild = 2 * i + 2;

	do {
		i = root;
		if (lChild < len && arr[lChild] > arr[root])
			root = lChild;

		if (rChild < len && arr[rChild] > arr[root])
			root = rChild;

		if (root != i) {
			tempNum = arr[i];
			arr[i] = arr[root];
			arr[root] = tempNum;
			lChild = 2 * root + 1;
			rChild = 2 * root + 2;
		}
	} while (root != i);
}

void heapSort(short arr[], short empty, short len) {
	short tempNum;
	len += 1;
	for (short i = len / 2 - 1; i >= 0; i--)
		heapify(arr, len, i);
	for (short i = len - 1; i > 0; i--) {
		tempNum = arr[0];
		arr[0] = arr[i];
```

```
            arr[i] = tempNum;
            heapify(arr, i, 0);
        }
}
```

Joonis 4-2 heapSort.cpp faili kood

Lisa 5 – Kiirsoortimise algoritmi kood

```
#ifndef quickSort_h
#define quickSort_h
short partition(short arr[], short start, short end_f);
void quickSort(short arr[], short start, short end_f);
#endif
```

Joonis 5-1 quickSort.h faili kood

```
#include "quickSort.h"

short partition(short arr[], short start, short end_f) {
    short pivot = arr[end_f];
    short tempNum;
    short i = (start - 1);

    for (short j = start; j <= end_f - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            tempNum = arr[i];
            arr[i] = arr[j];
            arr[j] = tempNum;
        }
    }
    i++;
    tempNum = arr[i];
    arr[i] = arr[end_f];
    arr[end_f] = tempNum;
    return i;
}

void quickSort(short arr[], short start, short end_f) {
    if (start >= end_f) return;
    short divider = partition(arr, start, end_f);
    quickSort(arr, start, divider - 1);
    quickSort(arr, divider + 1, end_f);
}
```

Joonis 5-2 quickSort.cpp faili kood

Lisa 6 – Mestimissoortimise algoritmi kood

```
#ifndef mergeSort_h
#define mergeSort_h
#include <stdlib.h>
void merge(short arr[], short iLeft, short iMiddle, short iRight);
void mergeSort(short arr[], short iLeft, short iRight);
#endif
```

Joonis 6-1 mergeSort.h faili kood

```
#include "mergeSort.h"

void merge(short arr[], short iLeft, short iMiddle, short iRight)
{
	short n1 = iMiddle - iLeft + 1;
	short n2 = iRight - iMiddle;

	short* LeftArr = (short*)calloc(n1, sizeof(short));
	short* RightArr = (short*)calloc(n2, sizeof(short));

	for (short i = 0; i < n1; i++) LeftArr[i] = arr[iLeft + i];
	for (short j = 0; j < n2; j++) RightArr[j] = arr[iMiddle + 1 + j];

	short i = 0, j = 0, k = iLeft;
	for (; i < n1 && j < n2; k++) {
		if (LeftArr[i] <= RightArr[j]) {
			arr[k] = LeftArr[i];
			i++;
		}
		else {
			arr[k] = RightArr[j];
			j++;
		}
	}

	for (; i < n1; i++, k++) arr[k] = LeftArr[i];

	for (; j < n2; j++, k++) arr[k] = RightArr[j];
	free(LeftArr);
	free(RightArr);
}


void mergeSort(short arr[], short iLeft, short iRight) {
	if (iLeft < iRight) {
```

```
            short iMiddle = (iLeft + iRight) / 2;

            mergeSort(arr, iLeft, iMiddle);
            mergeSort(arr, iMiddle + 1, iRight);

            merge(arr, iLeft, iMiddle, iRight);
        }
}
```

Joonis 6-2 mergeSort.cpp faili kood

Lisa 7 – Timsoortimise algoritmi kood

See sorteerimisalgoritm kasutab juba mainitud ja eelnevalt loodud sorteerimisalgoritme. (vt Lisa 3 ja Lisa 6)

```
#ifndef timSort_h
#define timSort_h
#include <insertionSort.h>
#include <mergeSort.h>
void timSort(short arr[], short empty, short n);
#endif
```

Joonis 7-1 timSort.h faili kood

```
#include "timSort.h"

short min(short a, short b) {
	if (a < b) return a;
	return b;
}

void timSort(short arr[], short empty, short n)
{
	short size, iLeft, iMiddle, iRight;
	const short run = 32;
	n += 1;
	for (short i = 0; i < n; i += run)
		insertionSort(arr, i, min((i + 31), (n - 1)));
	for (size = run; size < n; size = 2 * size)
	{
		for (iLeft = 0; iLeft < n; iLeft += 2 * size)
		{
			iMiddle = iLeft + size - 1;
			iRight = min((iLeft + 2 * size - 1), (n - 1));
			merge(arr, iLeft, iMiddle, iRight);
		}
	}
}
```

Joonis 7-2 timSort.cpp faili kood

Lisa 8 – Introsoortimise algoritmi kood

See sorteerimisalgoritm kasutab juba mainitud ja eelnevalt loodud sorteerimisalgoritme. (vt Lisa 3 ja Lisa 5)

```
#ifndef introSort_h
#define introSort_h
#include <insertionSort.h>
#include <math.h>
#include <quickSort.h>
void introSort(short arr[], short empty, short len);
#endif
```

Joonis 8-1 introSort.h faili kood

```
#include "introSort.h"

void _heapify(short arr[], short len, short i, short start)
{
	short root = i;
	short tempNum;
	short lChild = 2 * i + 1 - start;
	short rChild = 2 * i + 2 - start;
	do {
		i = root;
		if (lChild < len && arr[lChild] > arr[root])
			root = lChild;

		if (rChild < len && arr[rChild] > arr[root])
			root = rChild;

		if (root != i) {
			tempNum = arr[i];
			arr[i] = arr[root];
			arr[root] = tempNum;
			lChild = 2 * root + 1 - start;
			rChild = 2 * root + 2 - start;
		}
	} while (root != i);
}

void _heapSort(short arr[], short start, short end) {
	short tempNum, sum = start + end;
	for (short i = short(ceil(sum / 2.0)) - 1; i >= start; i--)
		_heapify(arr, end + 1, i, start);
	for (short i = end; i > start; i--) {
		tempNum = arr[start];
		arr[start] = arr[i];
		arr[i] = tempNum;
		_heapify(arr, i, start, start);
```

```
	}
}

void introsort_imp(short arr[], short start, short end, short depth) {

	if (end - start < 16) {
		insertionSort(arr, start, end);
		return;
	}

	if (depth == 0) {
		_heapSort(arr, start, end);
		return;
	}

	short divider = partition(arr, start, end);
	introsort_imp(arr, start, divider - 1, depth - 1);
	introsort_imp(arr, divider + 1, end, depth - 1);
}

void introSort(short arr[], short empty, short len) {
	len += 1;
	short depth = 2 * short(log(double(len - 1)));
	introsort_imp(arr, 0, len - 1, depth);
}
```

Joonis 8-2 introSort.cpp faili kood

Lisa 9 – Sorteerimisalgoritmide animeerimine

Kõiki sorteerimisalgoritmide animatsioone on võimalik vaadata saidil, mille lõi selle töö autor. [28]