**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-22 Присяжний А. О.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Ахаладзе І.Е.*

Київ 2023

Зміст

[1 Мета лабораторної роботи 3](#_Toc109342184)

[2 ЗаВдання 4](#_Toc109342185)

[3 Виконання 6](#_Toc109342186)

[3.1 Псевдокод алгоритму 6](#_Toc109342187)

[3.2 Програмна реалізація алгоритму 6](#_Toc109342188)

[3.2.1 Вихідний код 6](#_Toc109342189)

[Висновок 7](#_Toc109342190)

[Критерії оцінювання 8](#_Toc109342191)

# Мета лабораторної роботи

Мета роботи – вивчити основні алгоритми зовнішнього сортування та способи їх модифікації, оцінити поріг їх ефективності.

# Завдання

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм зовнішнього сортування за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування та відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі (розмір файлу має бути не менше 10 Мб, можна значно більше).

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

Рекомендується попередньо впорядкувати серії елементів довжиною, що займає не менше 100Мб або використати інші підходи для пришвидшення процесу сортування.

Зробити узагальнений висновок з лабораторної роботи, у якому порівняти базову та модифіковану програми. У висновку деталізувати, які саме модифікації було виконано і який ефект вони дали.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм сортування** |
| 1 | Пряме злиття |
| 2 | Природне (адаптивне) злиття |
| 3 | Збалансоване багатошляхове злиття |
| 4 | Багатофазне сортування |
| 5 | Пряме злиття |
| 6 | Природне (адаптивне) злиття |
| 7 | Збалансоване багатошляхове злиття |
| 8 | Багатофазне сортування |
| 9 | Пряме злиття |
| 10 | Природне (адаптивне) злиття |
| 11 | Збалансоване багатошляхове злиття |
| 12 | Багатофазне сортування |
| 13 | Пряме злиття |
| 14 | Природне (адаптивне) злиття |
| 15 | Збалансоване багатошляхове злиття |
| 16 | Багатофазне сортування |
| 17 | Пряме злиття |
| 18 | Природне (адаптивне) злиття |
| 19 | Збалансоване багатошляхове злиття |
| 20 | Багатофазне сортування |
| 21 | Пряме злиття |
| 22 | Природне (адаптивне) злиття |
| 23 | Збалансоване багатошляхове злиття |
| 24 | Багатофазне сортування |
| 25 | Пряме злиття |
| 26 | Природне (адаптивне) злиття |
| 27 | Збалансоване багатошляхове злиття |
| 28 | Багатофазне сортування |
| 29 | Пряме злиття |
| 30 | Природне (адаптивне) злиття |
| 31 | Збалансоване багатошляхове злиття |
| 32 | Багатофазне сортування |
| 33 | Пряме злиття |
| 34 | Природне (адаптивне) злиття |
| 35 | Збалансоване багатошляхове злиття |

# Виконання

## Псевдокод алгоритму

**int** index\_of\_file\_to\_write, ideal\_amount\_of\_series[amount\_of\_supporting\_files], amount\_of\_empty\_series[amount\_of\_supporting\_files], level;

**select\_supporting\_file\_to\_write\_series()** {

**int** max\_amount\_of\_series;

**if** (amount\_of\_empty\_series[index\_of\_file\_to\_write] < amount\_of\_empty\_series[index\_of\_file\_to\_write + 1]) **do**

index\_of\_file\_to\_write = index\_of\_file\_to\_write + 1;

**else** **do**

**if** (amount\_of\_empty\_series[index\_of\_file\_to\_write] == 0) **do**

level = level + 1;

max\_amount\_of\_series = ideal\_amount\_of\_series[0];

**for** i = 0 **to** amount\_of\_supporting\_files - 1 **do**

amount\_of\_empty\_series[i] = max\_ideal\_amount\_of\_series + ideal\_amount\_of\_series[i + 1] - ideal\_amount\_of\_series[i];

ideal\_amount\_of\_series[i] = max\_ideal\_amount\_of\_series + ideal\_amount\_of\_series[i + 1];

**end for**

**end if**

index\_of\_file\_to\_write = 0;

**end if**

amount\_of\_empty\_series[index\_of\_file\_to\_write] = amount\_of\_empty\_series[index\_of\_file\_to\_write] – 1;

}

**main()** {

**int** max\_amount\_of\_merged\_series, amount\_of\_active\_files, i, min, index\_of\_file\_with\_min\_element;

**fstream** file\_to\_sort;

**fstream** supporting\_files[amount\_of\_supporting\_files];

**int** supporting\_files\_indexes[amount\_of\_supporting\_files], active\_supporting\_files\_indexes[amount\_of\_supporting\_files-1];

**відкрити** file\_to\_sort на читання;

**for** i = 0 **to** amount\_of\_supporting\_files - 1 **do**

**відкрити** supporting\_files[i] на запис;

**end for**

**for** i = 0 **to** amount\_of\_supporting\_files - 1 **do**

ideal\_amount\_of\_series[i] = 1;

amount\_of\_empty\_series[i] = 0;

**записати** одну серію з file\_to\_sort в supporting\_files[i];

**end for**

level = 1;

index\_of\_file\_to\_write = 0;

**while** (не кінець file\_to\_sort) **do**

select\_supporting\_file\_to\_write\_series();

**if** (останній елемент supporting\_files[index\_of\_file\_to\_write] більший за перший елемент file\_to\_sort) **do**

**записати** одну серію з file\_to\_sort в supporting\_files[index\_of\_file\_to\_write];

**else**

**записати** одну серію з file\_to\_sort в supporting\_files[index\_of\_file\_to\_write];

**if** (file\_to\_sort не закінчився) **do**

**записати** одну серію з file\_to\_sort в supporting\_files[index\_of\_file\_to\_write];

**else**

amount\_of\_empty\_series[index\_of\_file\_to\_write] = amount\_of\_empty\_series[index\_of\_file\_to\_write] + 1;

**end if**

**end if**

**end while**

**for** i = 0 **to** amount\_of\_supporting\_files-1 **do**

supporting\_files\_indexes[i] = i;

**відкрити** supporting\_files[i] на читання;

**end for**

**відкрити** supporting\_files[amount\_of\_supporting\_files-1] на запис;

**do**

max\_amount\_of\_merged\_series = ideal\_amount\_of\_series[amount\_of\_supporting\_files - 2];

amount\_of\_empty\_series[amount\_of\_supporting\_files - 1] = 0;

**do**

amount\_of\_active\_files = 0

**for** i = 0 **to** amount\_of\_supporting\_files-1 **do**

**if** (amount\_of\_empty\_series[i] > 0) **do**

amount\_of\_empty\_series[i] = amount\_of\_empty\_series[i] – 1;

**else**

active\_supporting\_files\_indexes [amount\_of\_active\_files] = supporting\_files\_indexes[i];

amount\_of\_active\_files = amount\_of\_active\_files+1;

**end if**

**end for**

**if** (amount\_of\_active\_files == 0) **do**

amount\_of\_empty\_series[amount\_of\_supporting\_files-1] = amount\_of\_empty\_series[amount\_of\_supporting\_files-1]–1;

**else**

**do**

i = 0;

index\_of\_file\_with\_min\_element = 0;

min = перший елемент supporting\_files[active\_supporting\_files\_indexes[0]];

**while** (i < amount\_of\_active\_files) **do**

i = i + 1;

**int** current\_element = перший елемент supporting\_files[active\_supporting\_files\_indexes[i]];

**if** (current\_element < min) **do**

min = current\_element;

index\_of\_file\_with\_min\_element = i;

**end if**

**end while**

скопіювати елемент з supporting\_files[active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]] в supporting\_files[supporting\_files\_indexes[amount\_of\_supporting\_files-1]];

**if** (кінець серії supporting\_files[active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]]) **do**

amount\_of\_active\_files = amount\_of\_active\_files – 1;active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]] = supporting\_files\_indexes[amount\_of\_active\_files];

**end if**

**while** (amount\_of\_active\_files > 0)

max\_amount\_of\_merged\_series = max\_amount\_of\_merged\_series – 1;

**end if**

**while** (max\_amount\_of\_merged\_series > 0)

**закрити** supporting\_files[supporting\_files\_indexes[amount\_of\_supporting\_files-1]];

**відкрити** supporting\_files[supporting\_files\_indexes[amount\_of\_supporting\_files-1]] на читання;

**int** last\_supporting\_file\_index = supporting\_files\_indexes[amount\_of\_supporting\_files - 1];

**int** amount\_of\_empty\_series\_in\_last\_file = amount\_of\_empty\_series[amount\_of\_supporting\_files - 1];

max\_amount\_of\_merged\_series = ideal\_amount\_of\_series[amount\_of\_supporting\_files - 2];

**for** i = amount\_of\_supporting\_files - 1 **to** 1 **do**

supporting\_files\_indexes[i] = supporting\_files\_indexes[i - 1];

amount\_of\_empty\_series[i] = amount\_of\_empty\_series[i - 1];

ideal\_amount\_of\_series[i] = ideal\_amount\_of\_series[i - 1] - max\_amount\_of\_merged\_series;

**end for**

supporting\_files\_indexes[0] = last\_supporting\_file\_index;

amount\_of\_empty\_series[0] = amount\_of\_empty\_series\_in\_last\_file;

ideal\_amount\_of\_series[0] = max\_amount\_of\_merged\_series;

level = level – 1;

**while** (level > 0)

//відсортований файл зберігається у supporting\_files[supporting\_files\_indexes[0]]

}

## Програмна реалізація алгоритму

### Вихідний код

**ArrayReader.h**

#pragma once

#include "Reader.h"

class ArrayReader : public Reader {

public:

ArrayReader(int\* array);

int read() override;

void read\_range(int\*& dest\_begin, int\*& dest\_end, int size\_in\_mb) override;

void open\_from\_beggining() override;

int peek() override;

~ArrayReader();

};

**FileCreator.h**

#pragma once

#include <string>

#include <fstream>

using namespace std;

class FileCreator {

const int size\_of\_initial\_file\_in\_mb;

const wchar\_t\* initial\_file\_path;

public:

FileCreator(const wchar\_t\* initial\_file\_path, int size\_of\_initial\_file\_in\_mb);

void create\_initial\_file() const;

const wchar\_t\* get\_initial\_file\_path() const;

};

**FileMapping.h**

#pragma once

#include <algorithm>

#include "Reader.h"

using namespace std;

class FileMapping : public Reader {

const wchar\_t\* file\_path;

DWORD memory\_allocation\_granularity;

int part\_size\_in\_granularity;

HANDLE file;

DWORD part\_size;

HANDLE file\_mapping;

LPVOID mapped\_data;

DWORD view\_access;

bool is\_open;

int part\_counter;

private:

void open(DWORD creation\_access, DWORD creation\_disposition, DWORD protect, DWORD view\_access, int part\_size\_in\_granularity);

void switch\_part();

public:

FileMapping(const wchar\_t\* file\_path);

int read() override;

void read\_range(int\*& dest\_begin, int\*& dest\_end, int size\_in\_mb) override;

int peek() override;

void open\_from\_beggining() override;

void close();

~FileMapping();

};

**FileSorter.h**

#pragma once

#include <string>

#include <fstream>

#include "FileMapping.h"

#include "ArrayReader.h"

#include <iostream>

class FileSorter {

const int amount\_of\_supporting\_files;

Reader\* data\_to\_sort;

string\* supporting\_files\_names;

string result\_file\_name;

int \*active\_supporting\_files\_indexes, \*supporting\_files\_names\_indexes,

\*ideal\_amount\_of\_series, \*amount\_of\_empty\_series, \*first\_numbers;

int level;

DWORD memory\_allocation\_granularity;

int part\_size\_in\_granularity;

private:

void init\_memory\_granularity();

void create\_supporting\_files\_names(string prefix, string extension);

void init\_files\_data();

void pre\_sort();

void make\_initial\_spliting();

void select\_supporting\_file\_to\_write\_series(int& index\_of\_file\_to\_write);

int write\_series(Reader\* from, ofstream& destination);

int calculate\_amount\_of\_active\_files();

void shift\_suppotring\_files\_indexes();

void rename\_sorted\_file();

void delete\_supporting\_files();

void merge\_one\_serie(int amount\_of\_active\_files, fstream\* active\_supporting\_files);

void merge\_series\_from\_level(fstream\* active\_supporting\_files);

public:

FileSorter(Reader\* data, string file\_extension, string supporting\_file\_prefix, int amount\_of\_supporting\_files, string result\_file\_name);

void polyphase\_merge\_sort();

void display\_sorted\_file();

~FileSorter();

};

**Reader.h**

#pragma once

#include <iostream>

#include <Windows.h>

using namespace std;

class Reader

{

protected:

DWORD data\_size;

DWORD recorded\_data\_size;

int\* data;

public:

virtual int read() = 0;

virtual void read\_range(int\*& dest\_begin, int\*& dest\_end, int size\_in\_mb) = 0;

bool is\_end() const;

virtual int peek() = 0;

virtual void open\_from\_beggining() = 0;

};

**ArrayReader.cpp**

#include "ArrayReader.h"

ArrayReader::ArrayReader(int\* array) {

data = array;

recorded\_data\_size = 0;

data\_size = sizeof(data);

}

int ArrayReader::read() {

int number = data[recorded\_data\_size / sizeof(int)];

recorded\_data\_size += sizeof(int);

return number;

}

void ArrayReader::read\_range(int\*& dest\_begin, int\*& dest\_end, int size\_in\_mb) {

DWORD size\_in\_bytes = size\_in\_mb \* 1024 \* 1024;

dest\_begin = data + (recorded\_data\_size / sizeof(int));

DWORD actual\_size\_of\_data;

if (recorded\_data\_size + size\_in\_bytes > data\_size)

actual\_size\_of\_data = (data\_size - recorded\_data\_size) / sizeof(int);

else

actual\_size\_of\_data = size\_in\_bytes / sizeof(int);

dest\_end = data + actual\_size\_of\_data;

recorded\_data\_size += actual\_size\_of\_data \* sizeof(int);

}

int ArrayReader::peek() {

if (is\_end())

return 0;

return data[recorded\_data\_size/ sizeof(int)];

}

void ArrayReader::open\_from\_beggining() {

recorded\_data\_size = 0;

}

ArrayReader::~ArrayReader() {

delete[] data;

}

**FileCreator.cpp**

#include "FileCreator.h"

FileCreator::FileCreator(const wchar\_t\* initial\_file\_path, int size\_of\_initial\_file\_in\_mb) : size\_of\_initial\_file\_in\_mb(size\_of\_initial\_file\_in\_mb) {

this->initial\_file\_path = initial\_file\_path;

}

void FileCreator::create\_initial\_file() const{

srand(time(nullptr));

ofstream initial\_file(initial\_file\_path, ios::binary);

if (!initial\_file)

throw "Can't create initial file!";

int amount\_of\_numbers = size\_of\_initial\_file\_in\_mb \* 1024 \* 1024 / sizeof(int);

for (int i = 0; i < amount\_of\_numbers; ++i) {

int random\_number = rand();

initial\_file.write((char\*)&random\_number, sizeof(int));

}

initial\_file.close();

}

const wchar\_t\* FileCreator::get\_initial\_file\_path() const {

return initial\_file\_path;

}

**FileMapping.cpp**

#include "FileMapping.h"

FileMapping::FileMapping(const wchar\_t\* file\_path) : file\_path(file\_path) {

\_SYSTEM\_INFO s;

GetSystemInfo(&s);

memory\_allocation\_granularity = s.dwAllocationGranularity;

part\_size\_in\_granularity = 10 \* 1024 \* 1024 / memory\_allocation\_granularity;

open(GENERIC\_READ | GENERIC\_WRITE, OPEN\_ALWAYS, PAGE\_READWRITE, FILE\_MAP\_READ | FILE\_MAP\_WRITE, part\_size\_in\_granularity);

}

void FileMapping::open(DWORD creation\_access, DWORD creation\_disposition, DWORD protect, DWORD view\_access, int part\_size\_in\_granularity) {

file = CreateFile(file\_path, creation\_access, 0, NULL, creation\_disposition, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (file == INVALID\_HANDLE\_VALUE) {

throw "File Creation error";

}

data\_size = GetFileSize(file, NULL);

part\_size = part\_size\_in\_granularity \* memory\_allocation\_granularity;

file\_mapping = CreateFileMapping(file, NULL, protect, 0, 0, NULL);

if (file\_mapping == NULL) {

CloseHandle(file);

throw "File Mapping Creation error";

}

mapped\_data = MapViewOfFile(file\_mapping, view\_access, 0, 0, (part\_size > data\_size) ? data\_size : part\_size);

recorded\_data\_size = 0;

part\_counter = 1;

this->view\_access = view\_access;

if (mapped\_data == NULL) {

CloseHandle(file\_mapping);

CloseHandle(file);

throw "Map View error";

}

data = static\_cast<int\*>(mapped\_data);

is\_open = true;

}

void FileMapping::open\_from\_beggining() {

UnmapViewOfFile(mapped\_data);

mapped\_data = MapViewOfFile(file\_mapping, view\_access, 0, 0, (part\_size > data\_size) ? data\_size : part\_size);

recorded\_data\_size = 0;

part\_counter = 1;

this->view\_access = view\_access;

if (mapped\_data == NULL) {

CloseHandle(file\_mapping);

CloseHandle(file);

throw "Map View error";

}

data = static\_cast<int\*>(mapped\_data);

}

int FileMapping::read() {

int number = data[(recorded\_data\_size - part\_size \* (part\_counter - 1)) / sizeof(int)];

recorded\_data\_size += sizeof(int);

if (recorded\_data\_size % part\_size == 0)

switch\_part();

return number;

}

void FileMapping::read\_range(int\*& dest\_begin, int\*& dest\_end, int size\_in\_mb) {

DWORD size\_in\_bytes = size\_in\_mb \* 1024 \* 1024;

dest\_begin = data + ((recorded\_data\_size - part\_size \* (part\_counter - 1)) / sizeof(int));

DWORD actual\_size\_of\_data;

if (recorded\_data\_size + size\_in\_bytes > data\_size)

actual\_size\_of\_data = (data\_size - recorded\_data\_size) / sizeof(int);

else if (recorded\_data\_size + size\_in\_bytes > part\_size \* part\_counter)

actual\_size\_of\_data = (part\_size \* part\_counter - recorded\_data\_size) / sizeof(int);

else

actual\_size\_of\_data = size\_in\_bytes / sizeof(int);

dest\_end = data + actual\_size\_of\_data;

recorded\_data\_size += actual\_size\_of\_data \* sizeof(int);

if (recorded\_data\_size % part\_size == 0)

switch\_part();

}

int FileMapping::peek() {

if (is\_end())

return 0;

return data[(recorded\_data\_size - part\_size \* (part\_counter - 1)) / sizeof(int)];

}

void FileMapping::switch\_part() {

if (is\_end())

return;

LPVOID mapped\_data = MapViewOfFile(file\_mapping, view\_access, 0, part\_counter \* part\_size, (part\_size > data\_size - part\_size \* part\_counter ? data\_size - part\_size \* part\_counter : part\_size));

if (mapped\_data == NULL) {

CloseHandle(file\_mapping);

CloseHandle(file);

throw "Map View error";

}

data = static\_cast<int\*>(mapped\_data);

part\_counter++;

}

void FileMapping::close() {

if (!is\_open)

return;

CloseHandle(file);

CloseHandle(file\_mapping);

UnmapViewOfFile(mapped\_data);

is\_open = false;

}

FileMapping::~FileMapping() {

close();

}

**FileSorter.cpp**

#include "FileSorter.h"

FileSorter::FileSorter(Reader\* data, string file\_extension, string supporting\_file\_prefix, int amount\_of\_supporting\_files, string result\_file\_name)

: amount\_of\_supporting\_files(amount\_of\_supporting\_files), result\_file\_name(result\_file\_name) {

data\_to\_sort = data;

init\_memory\_granularity();

create\_supporting\_files\_names(supporting\_file\_prefix, file\_extension);

init\_files\_data();

level = 1;

remove(result\_file\_name.c\_str());

}

void FileSorter::init\_memory\_granularity() {

\_SYSTEM\_INFO s;

GetSystemInfo(&s);

memory\_allocation\_granularity = s.dwAllocationGranularity;

part\_size\_in\_granularity = 10 \* 1024 \* 1024 / memory\_allocation\_granularity;

}

void FileSorter::create\_supporting\_files\_names(string prefix, string extension) {

supporting\_files\_names = new string[amount\_of\_supporting\_files];

for (int i = 0; i < amount\_of\_supporting\_files; ++i)

supporting\_files\_names[i] = prefix + to\_string(i + 1) + extension;

}

void FileSorter::init\_files\_data() {

supporting\_files\_names\_indexes = new int[amount\_of\_supporting\_files];

ideal\_amount\_of\_series = new int[amount\_of\_supporting\_files];

amount\_of\_empty\_series = new int[amount\_of\_supporting\_files];

first\_numbers = new int[amount\_of\_supporting\_files];

active\_supporting\_files\_indexes = new int[amount\_of\_supporting\_files - 1];

for (int i = 0; i < amount\_of\_supporting\_files - 1; ++i) {

ideal\_amount\_of\_series[i] = 1;

amount\_of\_empty\_series[i] = 1;

supporting\_files\_names\_indexes[i] = i;

}

ideal\_amount\_of\_series[amount\_of\_supporting\_files - 1] = 0;

amount\_of\_empty\_series[amount\_of\_supporting\_files - 1] = 0;

supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1] = amount\_of\_supporting\_files - 1;

}

void FileSorter::pre\_sort() {

const int part\_size\_in\_mb = 10;

while (!data\_to\_sort->is\_end()) {

int \*begin, \*end;

data\_to\_sort->read\_range(begin, end, part\_size\_in\_mb);

sort(begin, end);

}

data\_to\_sort->open\_from\_beggining();

}

void FileSorter::make\_initial\_spliting() {

ofstream\* supporting\_files = new ofstream[amount\_of\_supporting\_files - 1];

int\* last\_recorded\_number = new int[amount\_of\_supporting\_files - 1];

for (int i = 0; i < amount\_of\_supporting\_files - 1 && !data\_to\_sort->is\_end(); ++i) {

supporting\_files[i].open(supporting\_files\_names[i], ios::binary);

last\_recorded\_number[i] = write\_series(data\_to\_sort, supporting\_files[i]);

--amount\_of\_empty\_series[i];

}

int index\_of\_file\_to\_write = 0;

while (!data\_to\_sort->is\_end()) {

select\_supporting\_file\_to\_write\_series(index\_of\_file\_to\_write);

bool is\_series\_concat = data\_to\_sort->peek() > last\_recorded\_number[index\_of\_file\_to\_write];

last\_recorded\_number[index\_of\_file\_to\_write] = write\_series(data\_to\_sort, supporting\_files[index\_of\_file\_to\_write]);

if (!is\_series\_concat)

continue;

if (!data\_to\_sort->is\_end())

last\_recorded\_number[index\_of\_file\_to\_write] = write\_series(data\_to\_sort, supporting\_files[index\_of\_file\_to\_write]);

else

++amount\_of\_empty\_series[index\_of\_file\_to\_write];

}

for (int i = 0; i < amount\_of\_supporting\_files - 1; ++i)

supporting\_files[i].close();

delete[] supporting\_files;

delete[] last\_recorded\_number;

}

void FileSorter::select\_supporting\_file\_to\_write\_series(int& index\_of\_file\_to\_write) {

int max\_ideal\_amount\_of\_series;

if (amount\_of\_empty\_series[index\_of\_file\_to\_write] < amount\_of\_empty\_series[index\_of\_file\_to\_write + 1]) {

index\_of\_file\_to\_write++;

--amount\_of\_empty\_series[index\_of\_file\_to\_write];

return;

}

if (amount\_of\_empty\_series[index\_of\_file\_to\_write] == 0) {

++level;

max\_ideal\_amount\_of\_series = ideal\_amount\_of\_series[0];

for (int i = 0; i < amount\_of\_supporting\_files - 1; ++i) {

amount\_of\_empty\_series[i] = max\_ideal\_amount\_of\_series + ideal\_amount\_of\_series[i + 1] - ideal\_amount\_of\_series[i];

ideal\_amount\_of\_series[i] = max\_ideal\_amount\_of\_series + ideal\_amount\_of\_series[i + 1];

}

}

index\_of\_file\_to\_write = 0;

--amount\_of\_empty\_series[index\_of\_file\_to\_write];

}

int FileSorter::write\_series(Reader\* from, ofstream& destination) {

int current\_number;

do {

current\_number = from->read();

destination.write((char\*)&current\_number, sizeof(current\_number));

} while (from->peek() >= current\_number && !from->is\_end());

return current\_number; //last recorded number

}

void FileSorter::polyphase\_merge\_sort() {

pre\_sort();

make\_initial\_spliting();

fstream\* active\_supporting\_files = new fstream[amount\_of\_supporting\_files];

for (int i = 0; i < amount\_of\_supporting\_files - 1; ++i) {

active\_supporting\_files[i].open(supporting\_files\_names[i], ios::in | ios::binary);

active\_supporting\_files[i].read((char\*) (first\_numbers + i), sizeof(int));

}

do {

merge\_series\_from\_level(active\_supporting\_files);

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]].close();

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]].open(supporting\_files\_names[supporting\_files\_names\_indexes[amount\_of\_supporting\_files-1]], ios::in | ios::binary);

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]].read((char\*) (first\_numbers + supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]), sizeof(int));

shift\_suppotring\_files\_indexes();

--level;

} while (level > 0);

for (int i = 0; i < amount\_of\_supporting\_files; ++i)

active\_supporting\_files[i].close();

rename\_sorted\_file();

delete\_supporting\_files();

delete[] active\_supporting\_files;

}

void FileSorter::merge\_series\_from\_level(fstream\* active\_supporting\_files) {

int max\_amount\_of\_merged\_series = ideal\_amount\_of\_series[amount\_of\_supporting\_files - 2];

amount\_of\_empty\_series[amount\_of\_supporting\_files - 1] = 0;

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]].close();

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]].open(supporting\_files\_names[supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1]], ios::out | ios::trunc | ios::binary);

do {

int amount\_of\_active\_files = calculate\_amount\_of\_active\_files();

if (amount\_of\_active\_files == 0)

++amount\_of\_empty\_series[amount\_of\_supporting\_files - 1];

else

merge\_one\_serie(amount\_of\_active\_files, active\_supporting\_files);

--max\_amount\_of\_merged\_series;

} while (max\_amount\_of\_merged\_series > 0);

}

int FileSorter::calculate\_amount\_of\_active\_files() {

int amount\_of\_active\_files = 0;

for (int i = 0; i < amount\_of\_supporting\_files - 1; ++i) {

if (amount\_of\_empty\_series[i] > 0)

--amount\_of\_empty\_series[i];

else {

active\_supporting\_files\_indexes[amount\_of\_active\_files] = supporting\_files\_names\_indexes[i];

++amount\_of\_active\_files;

}

}

return amount\_of\_active\_files;

}

void FileSorter::shift\_suppotring\_files\_indexes() {

int last\_supporting\_file\_name\_index = supporting\_files\_names\_indexes[amount\_of\_supporting\_files - 1];

int amount\_of\_empty\_series\_in\_last\_file = amount\_of\_empty\_series[amount\_of\_supporting\_files - 1];

int max\_amount\_of\_merged\_series = ideal\_amount\_of\_series[amount\_of\_supporting\_files - 2];

for (int i = amount\_of\_supporting\_files - 1; i > 0; --i) {

supporting\_files\_names\_indexes[i] = supporting\_files\_names\_indexes[i - 1];

amount\_of\_empty\_series[i] = amount\_of\_empty\_series[i - 1];

ideal\_amount\_of\_series[i] = ideal\_amount\_of\_series[i - 1] - max\_amount\_of\_merged\_series;

}

supporting\_files\_names\_indexes[0] = last\_supporting\_file\_name\_index;

amount\_of\_empty\_series[0] = amount\_of\_empty\_series\_in\_last\_file;

ideal\_amount\_of\_series[0] = max\_amount\_of\_merged\_series;

}

void FileSorter::rename\_sorted\_file() {

rename(supporting\_files\_names[supporting\_files\_names\_indexes[0]].c\_str(), result\_file\_name.c\_str());

}

void FileSorter::delete\_supporting\_files() {

for (int i = 0; i < amount\_of\_supporting\_files; i++)

remove(supporting\_files\_names[i].c\_str());

}

void FileSorter::merge\_one\_serie(int amount\_of\_active\_files, fstream\* active\_supporting\_files) {

do {

int i = 0, index\_of\_file\_with\_min\_element = 0, min = first\_numbers[active\_supporting\_files\_indexes[0]];

while (i < amount\_of\_active\_files - 1) {

++i;

int current\_element = first\_numbers[active\_supporting\_files\_indexes[i]];

if (current\_element < min) {

min = current\_element;

index\_of\_file\_with\_min\_element = i;

}

}

active\_supporting\_files[active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]].read((char\*) (first\_numbers + active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]), sizeof(int));

active\_supporting\_files[supporting\_files\_names\_indexes[amount\_of\_supporting\_files-1]].write((char\*)&min, sizeof(min));

if (first\_numbers[active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]] < min || active\_supporting\_files[active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element]].eof()) {

--amount\_of\_active\_files;

active\_supporting\_files\_indexes[index\_of\_file\_with\_min\_element] = active\_supporting\_files\_indexes[amount\_of\_active\_files];

}

} while (amount\_of\_active\_files > 0);

}

void FileSorter::display\_sorted\_file() {

ifstream file(result\_file\_name, ios::binary);

const int amount\_of\_displayed\_numbers = 5;

for (int i = 0; i < amount\_of\_displayed\_numbers; ++i) {

int number;

file.read((char\*)&number, sizeof(number));

cout << number << " ";

}

file.seekg(-1 \* sizeof(int) \* amount\_of\_displayed\_numbers, ios::end);

cout << "... ";

for (int i = 0; i < amount\_of\_displayed\_numbers; ++i) {

int number;

file.read((char\*)&number, sizeof(number));

cout << number << " ";

}

cout << endl;

file.close();

}

FileSorter::~FileSorter() {

delete[] supporting\_files\_names;

delete[] ideal\_amount\_of\_series;

delete[] amount\_of\_empty\_series;

delete[] active\_supporting\_files\_indexes;

delete[] supporting\_files\_names\_indexes;

delete[] first\_numbers;

}

**Reader.cpp**

#include "Reader.h"

bool Reader::is\_end() const {

return recorded\_data\_size >= data\_size;

}

**Lab1.cpp**

#include "FileCreator.h"

#include "FileSorter.h"

#include <ctime>

#include <iomanip>

int input\_positive\_number() {

int number;

char ch;

bool repeat;

do {

repeat = false;

if (!(cin >> number) || ((ch = getchar()) != '\n') || (number <= 0) || (number > INT32\_MAX)) {

cout << "Invalid data, input integer number from 1 to "<<INT32\_MAX<<" please." << endl;

cin.clear();

cin.ignore(100, '\n');

repeat = true;

}

} while (repeat);

return number;

}

int main() {

const int AMOUNT\_OF\_SUPPORTING\_FILES = 3;

string file\_extension = ".bin";

string supporting\_file\_prefix = "supporting\_";

string result\_file\_path = "result" + file\_extension;

const wchar\_t\* initial\_file\_path = L"initial.bin";

int repeat;

do {

cout << "Enter size of file to sort in megabytes:" << endl;

int size\_of\_initial\_file\_in\_mb = input\_positive\_number();

try {

FileCreator creator(initial\_file\_path, size\_of\_initial\_file\_in\_mb);

clock\_t beginning\_of\_creation, end\_of\_creation,

beginning\_of\_sorting, end\_of\_sorting;

beginning\_of\_creation = clock();

creator.create\_initial\_file();

end\_of\_creation = clock();

cout << "Creation of the file took " << fixed << double(end\_of\_creation - beginning\_of\_creation) / CLOCKS\_PER\_SEC << setprecision(5) << " seconds" << endl;

FileMapping file\_to\_sort(creator.get\_initial\_file\_path());

FileSorter sorter(&file\_to\_sort, file\_extension, supporting\_file\_prefix, AMOUNT\_OF\_SUPPORTING\_FILES, result\_file\_path);

beginning\_of\_sorting = clock();

sorter.polyphase\_merge\_sort();

end\_of\_sorting = clock();

cout << "Sorting of the file took " << fixed << double(end\_of\_sorting - beginning\_of\_sorting) / CLOCKS\_PER\_SEC << setprecision(5) << " seconds" << endl;

sorter.display\_sorted\_file();

cout << "Enter 1 to run program again or any other positive value from 2 to "<< INT32\_MAX<<" to stop it : " << endl;

repeat = input\_positive\_number();

file\_to\_sort.close();

}

catch (const char\* err) {

cout << err << endl;

repeat = 0;

}

} while (repeat == 1);

return 0;

}

**Lab1\_tests.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "../Lab1/FileCreator.h"

#include "../Lab1/FileSorter.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lab1tests

{

TEST\_CLASS(FileCreatorTest)

{

public:

TEST\_METHOD(creatre\_initial\_file\_creation\_test)

{

string file\_path = "creating\_test.bin";

wstring file\_path\_w = wstring(file\_path.begin(), file\_path.end());

const wchar\_t\* result\_path = file\_path\_w.c\_str();

const int size\_of\_file\_in\_mb = 1;

FileCreator creator(result\_path, size\_of\_file\_in\_mb);

creator.create\_initial\_file();

ifstream created\_file(file\_path, ios::binary);

Assert::IsTrue(created\_file.is\_open());

created\_file.close();

remove(file\_path.c\_str());

}

TEST\_METHOD(creatre\_initial\_file\_size\_test)

{

string file\_path = "creating\_test.bin";

wstring file\_path\_w = wstring(file\_path.begin(), file\_path.end());

const wchar\_t\* result\_path = file\_path\_w.c\_str();

const int size\_of\_file\_in\_mb = 1;

FileCreator creator(result\_path, size\_of\_file\_in\_mb);

creator.create\_initial\_file();

ifstream created\_file(file\_path, ios::binary);

created\_file.seekg(0, ios::end);

int file\_size = created\_file.tellg();

Assert::AreEqual(file\_size, (size\_of\_file\_in\_mb \* 1024 \* 1024));

created\_file.close();

remove(file\_path.c\_str());

}

};

TEST\_CLASS(FileSorterTest)

{

public:

TEST\_METHOD(polyphase\_merge\_sort\_test\_asc)

{

const int AMOUNT\_OF\_NUMBERS = 10;

const int AMOUNT\_OF\_SUPPORTING\_FILES = 3;

string file\_extension = ".bin";

string supporting\_file\_prefix = "test\_supporting\_";

string result\_file\_path = "test\_result" + file\_extension;

int\* data{ new int[AMOUNT\_OF\_NUMBERS] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} };

ArrayReader data\_to\_sort(data);

FileSorter sorter(&data\_to\_sort, file\_extension, supporting\_file\_prefix, AMOUNT\_OF\_SUPPORTING\_FILES, result\_file\_path);

sorter.polyphase\_merge\_sort();

ifstream file\_after\_sorting(result\_file\_path, ios::binary);

int current\_number, previous\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

while (!file\_after\_sorting.eof()) {

previous\_number = current\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

Assert::IsTrue(current\_number >= previous\_number);

}

file\_after\_sorting.close();

remove(result\_file\_path.c\_str());

}

TEST\_METHOD(polyphase\_merge\_sort\_test\_desc)

{

const int AMOUNT\_OF\_NUMBERS = 10;

const int AMOUNT\_OF\_SUPPORTING\_FILES = 3;

string file\_extension = ".bin";

string supporting\_file\_prefix = "test\_supporting\_";

string result\_file\_path = "test\_result" + file\_extension;

int\* data{ new int[AMOUNT\_OF\_NUMBERS] {10, 9, 8, 7, 6, 5, 4, 3, 2, 1} };

ArrayReader data\_to\_sort(data);

FileSorter sorter(&data\_to\_sort, file\_extension, supporting\_file\_prefix, AMOUNT\_OF\_SUPPORTING\_FILES, result\_file\_path);

sorter.polyphase\_merge\_sort();

ifstream file\_after\_sorting(result\_file\_path, ios::binary);

int current\_number, previous\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

while (!file\_after\_sorting.eof()) {

previous\_number = current\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

Assert::IsTrue(current\_number >= previous\_number);

}

file\_after\_sorting.close();

remove(result\_file\_path.c\_str());

}

TEST\_METHOD(polyphase\_merge\_sort\_test\_rand)

{

const int AMOUNT\_OF\_NUMBERS = 10;

const int AMOUNT\_OF\_SUPPORTING\_FILES = 3;

string file\_extension = ".bin";

string supporting\_file\_prefix = "test\_supporting\_";

string result\_file\_path = "test\_result" + file\_extension;

int\* data{ new int[AMOUNT\_OF\_NUMBERS] {5, 3, 8, 1, 2, 6, 10, 7, 4, 9} };

ArrayReader data\_to\_sort(data);

FileSorter sorter(&data\_to\_sort, file\_extension, supporting\_file\_prefix, AMOUNT\_OF\_SUPPORTING\_FILES, result\_file\_path);

sorter.polyphase\_merge\_sort();

ifstream file\_after\_sorting(result\_file\_path, ios::binary);

int current\_number, previous\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

while (!file\_after\_sorting.eof()) {

previous\_number = current\_number;

file\_after\_sorting.read((char\*)&current\_number, sizeof(current\_number));

Assert::IsTrue(current\_number >= previous\_number);

}

file\_after\_sorting.close();

remove(result\_file\_path.c\_str());

}

};

}

Висновок

При виконанні даної лабораторної роботи я реалізував алгоритм багатофазного сортування злиттям на мові програмування С++, а також зобразив його за допомогою псевдокоду. Звичайний алгоритм сортує файл розміром 20 Мб за 10 секунд, про що свідчить наступний скріншот виконання програми:

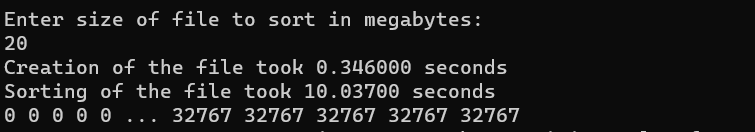


Рисунок 1 – Сортування файлу розміром 20 мегабайт звичайним алгоритмом багатофазного сортування

Щоб покращити швидкодію алгоритму я використав наступний підхід: для читання даних з початкового файлу я використовую технологію відображення файлу в оперативну пам’ять (File Mapping), розбиваю цей файл на частини розміром 10 Мб та сортую їх окремо за допомогою внутрішнього швидкого сортування (quick sort). File Mapping дозволяє багатократно пришвидшити операції читання і запису у файл, тому зменшується і загальний час роботи алгоритму сортування. Також, так як файл уже відсортований частинами, то у ньому значно менша кількість серій, ніж у невідсортованому, тому потрібно здійснювати менше операцій злиття під час сортування, що також пришвидшує роботу програми. У результаті модифікована версія алгоритму сортує файл на 20 Мб за 1 секунду, що у 10 разів швидше. Про це свідчить наступний скріншот:

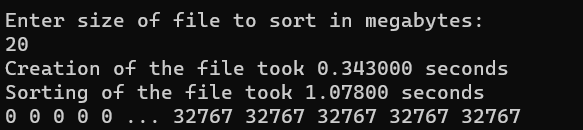


Рисунок 2 - Сортування файлу розміром 20 мегабайт модифікованим алгоритмом багатофазного сортування

Файл розміром 1 Гб модифікована версія алгоритму сортує за 3 хвилини 16 секунд (Рисунок 3).

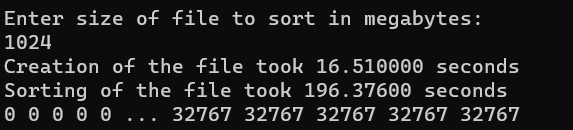


Рисунок 3 - Сортування файлу розміром 1 гігабайт модифікованим алгоритмом багатофазного сортування

Критерії оцінювання

У випадку здачі лабораторної роботи до 08.10.2022 включно максимальний бал дорівнює – 5. Після 08.10.2022 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 20%;
* програмна реалізація модифікацій – 20%;
* робота з git – 40%;
* висновок – 5%.