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) *ІП-22 Присяжний А. О.* ___________

(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І.Е.* ___________

(прізвище, ім'я, по батькові)

Київ 2023

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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

3 ВИКОНАННЯ

3.1 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]]

}
```

3.2 Програмна реалізація алгоритму

3.2.1 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_supportin
g_files - 1]].close();

    active_supporting_files[supporting_files_names_indexes[amount_of_supportin
g_files -
1]].open(supporting_files_names[supporting_files_names_indexes[amount_of_suppo
rting_files-1]], ios::in | ios::binary);

    active_supporting_files[supporting_files_names_indexes[amount_of_supportin
g_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_s
tr(), 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_";
```

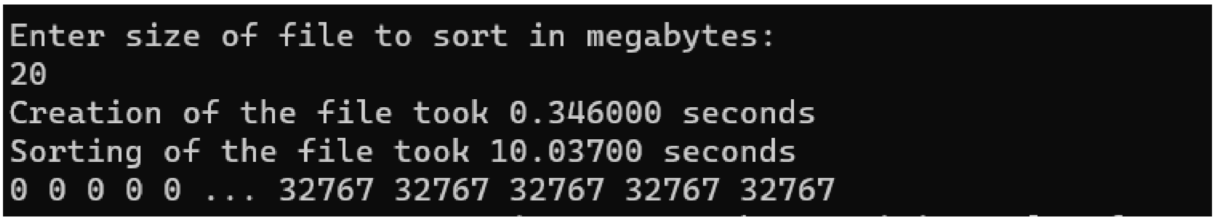
```
                string result_file_path = "test_result" + file_extension;

                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

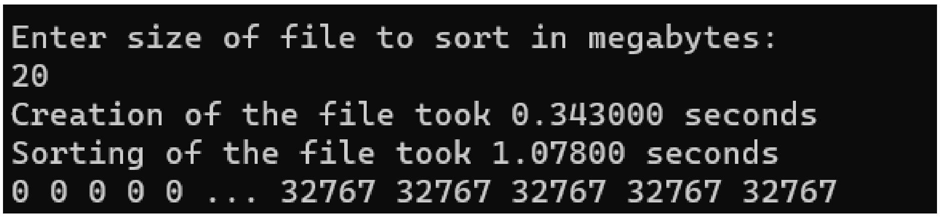
При виконанні даної лабораторної роботи я реалізував алгоритм багатофазного сортування злиттям на мові програмування C++, а також зобразив його за допомогою псевдокоду. Звичайний алгоритм сортує файл розміром 20 Мб за 10 секунд, про що свідчить наступний скріншот виконання програми:



```
Enter size of file to sort in megabytes:
20
Creation of the file took 0.346000 seconds
Sorting of the file took 10.03700 seconds
0 0 0 0 0 ... 32767 32767 32767 32767 32767
```

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

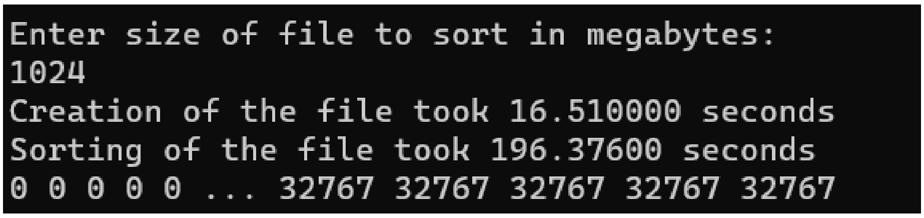
Щоб покращити швидкодію алгоритму я використав наступний підхід: для читання даних з початкового файлу я використовую технологію відображення файлу в оперативну пам’ять (File Mapping), розбиваю цей файл на частини розміром 10 Мб та сортую їх окремо за допомогою внутрішнього швидкого сортування (quick 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:



```
Enter size of file to sort in megabytes:
20
Creation of the file took 0.343000 seconds
Sorting of the file took 1.07800 seconds
0 0 0 0 0 ... 32767 32767 32767 32767 32767
```

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).



```
Enter size of file to sort in megabytes:
1024
Creation of the file took 16.510000 seconds
Sorting of the file took 196.37600 seconds
0 0 0 0 0 ... 32767 32767 32767 32767 32767
```

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%.