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-13 Пархомчук Ілля Вікторович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2022

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.

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

РозділенняНаФайли(Файл A, Файл B, Файл C)

1. Очистити файл C та B
2. flag = true;
3. Зчитати з фалу A old
4. Поки true

4.1. Якщо flag то

4.1.1. Обрати активним файлом для запису файл B

4.2. Інакше

4.2.1. Обрати активним файлом для запису файл C

4.3.Поки true

4.3.1. Записати у активний файл old

4.3.2. Зчитати з файлу A buf

4.3.3. Якщо кінець файлу A

4.3.3.1. Перервати цикл

4.3.4. Якщо old <= buf, то

4.3.4.1. old = buf

4.3.5. Інакше

4.3.5.1. old = buf

4.3.5.2. Перервати цикл

4.4. Якщо кінець фалу A

4.4.1. Повернути Розмір файлу C == 0

ЗлиттяФайлів(Файл A, Файл B, Файл C)

1. Зчитати old_B з файлу B
2. Зчитати old_C з файлу C
3. Поки true

3.1.Якщо old_B <= old_C

3.1.1. Записати old_B у файл A

3.1.2. Зчитати buf_B з файлу B

3.1.3. Якщо buf_B < old_B і не кінець фалу B

3.1.3.1. Поки true

3.1.3.1.1. Записати old_C у файл A

3.1.3.1.2. Зчитати buf_C з файлу C

3.1.3.1.3. Якщо кінець файлу C

3.1.3.1.3.1. Перервати цикл

3.1.3.1.4. Якщо old_C <= buf_C

3.1.3.1.4.1. old_C = buf_C

3.1.3.1.5. Інакше

3.1.3.1.5.1. old_C = buf_C

3.1.3.1.5.2. Перервати цикл

3.1.4. old_B = buf_B

3.2.Інакше

3.2.1. Повторити кроки з 3.1.1 по 3.1.4 замінивши old_B на old_C, buf_B на buf_C, файл B на файл C і навпаки.

3.3.Якщо кінець файлу B

3.3.1. Повторити

3.3.1.1. Записати old_C у файл A

3.3.1.2. Зчитати old_C з файлу C

3.3.2. Поки не кінець файлу C

3.3.3. Вихід з фукнції

3.4.Якщо кінець файлу C

3.4.1. Повторити кроки з 3.3.1 по 3.3.3 замінивши old_B на old_C, файл C на файл B

Сортування

1. Поки(не РозділенняНаФайли(Файл A, Файл B, Файл C))

1.1.ЗлиттяФайлів(Файл A, Файл B, Файл C)

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
#include <iostream>
#include <fstream>
#include <string>
#include <time.h>
#include <random>

#include <filesystem>

#include <assert.h>

using std::ofstream;
using std::ifstream;
using std::fstream;
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::string;


typedef int sort_type;

class Profiler
{
private:
	clock_t start;
	string message;

public:
	Profiler(string msg) : message(move(msg))
	{
		start = clock();
	}
	~Profiler()
	{
		cerr << "\nEnd " << message << " duration: " << (clock() - start) << " msec"
<< endl;
	}
};


bool Devide(fstream& A, fstream& B, fstream &C)
{
	using std::filesystem::resize_file;
	A.clear();
	B.clear();
	C.clear();

	resize_file("B.a", 0);
	resize_file("C.a", 0);

	A.seekg(0);
```

```
	B.seekp(0);
	C.seekp(0);

	bool flag = true;

	sort_type buf = 0, old = 0;

	A.read((char*)&old, sizeof(sort_type));
	while (true)
	{
		fstream& Active = flag ? B : C;
		flag = !flag;

		do
		{
			Active.write((char*)&old, sizeof(sort_type));

			A.read((char*)&buf, sizeof(sort_type));
			if (A.eof())
				break;
			if (old <= buf)
			{
				old = buf;
			}
			else
			{
				old = buf;
				break;
			}

		} while (true);
		if (A.eof())
		{
			A.clear();
			return (C.tellp() == 0);
		}
	}
}



void Merge(fstream& A, fstream& B, fstream& C)
{
	A.clear();
	B.clear();
	C.clear();
	A.seekg(0);
	B.seekp(0);
	C.seekp(0);


	sort_type old_B, old_C;
	B.read((char*)&old_B, sizeof(sort_type));
	C.read((char*)&old_C, sizeof(sort_type));
	sort_type buf_B, buf_C;


	while (true)
	{
		if (old_B <= old_C)
		{
			A.write((char*)&old_B, sizeof(sort_type));

			B.read((char*)&buf_B, sizeof(sort_type));
```

```
        if (buf_B < old_B && !B.eof())
        {
                do
                {
                        A.write((char*)&old_C, sizeof(sort_type));
                        C.read((char*)&buf_C, sizeof(sort_type));
                        if (C.eof()) break;
                        if (old_C <= buf_C)
                        {
                                old_C = buf_C;
                        }
                        else
                        {
                                old_C = buf_C;
                                break;
                        }
                } while (!C.eof());
        }
        old_B = buf_B;
}
else
{
        A.write((char*)&old_C, sizeof(sort_type));

        C.read((char*)&buf_C, sizeof(sort_type));

        if (buf_C < old_C && !C.eof())
        {
                do
                {
                        A.write((char*)&old_B, sizeof(sort_type));
                        B.read((char*)&buf_B, sizeof(sort_type));
                        if (C.eof())
                                break;
                        if (old_B <= buf_B)
                        {
                                old_B = buf_B;
                        }
                        else
                        {
                                old_B = buf_B;
                                break;
                        }
                } while (!B.eof());
        }
        old_C = buf_C;
}

if (B.eof())
{
        do
        {
                A.write((char*)&old_C, sizeof(sort_type));
                C.read((char*)&old_C, sizeof(sort_type));
        }while (!C.eof());

        return;
}
if (C.eof())
{
        do
        {
                A.write((char*)&old_B, sizeof(sort_type));
                B.read((char*)&old_B, sizeof(sort_type));
        } while (!B.eof());
```

```
                        return;
                }
        }
}

constexpr size_t SZ = 100*1024 * 1024 / sizeof(sort_type);

sort_type*  Generate(fstream& file)
{
        sort_type *buf = new sort_type[SZ];
        for (int i = 0; i < SZ; ++i)
        {
                buf[i] = rand() * rand();
        }
        file.write((char*)buf, SZ * sizeof(sort_type));
        return buf;
}

void Test(fstream& file)
{
        file.clear();
        file.seekg(0);
        sort_type buf;
        for (int i = 0; i <= SZ; ++i)
        {
                file.read((char*)&buf, sizeof(buf));
                cout << i << "op eof: " << std::boolalpha << file.eof() << '\n';
        }
}

void PrintFile(fstream& file)
{
        file.clear();
        file.seekg(0);
        sort_type buf;
        do
        {
                file.read((char*)&buf, sizeof(buf));
                if (file.eof())
                        break;
                cout << buf << ' ';
        } while (true);
        cout << endl;
}

void PrintArr(sort_type* A)
{
        for (int i = 0;i < SZ; )
        {
                cout << A[i] << ' ';
                ++i;
        }
        cout << '\n';
}

int main()
{


        srand(time(0));
        using std::ios_base;
        ofstream dummy("A.a"); dummy.close();
        dummy.open("B.a"); dummy.close();
        dummy.open("C.a"); dummy.close();
```

```
	fstream A("A.a", ios_base::binary | ios_base::in | ios_base::out);
	fstream B("B.a", ios_base::binary | ios_base::in | ios_base::out);
	fstream C("C.a", ios_base::binary | ios_base::in | ios_base::out);

	if (A.is_open() && B.is_open() && C.is_open())
	{
		cout << "all goood\n";
	}
	else
		return -1;

	auto cmp = Generate(A);
	{
		Profiler qs("std::sort");
		std::sort(cmp, cmp + SZ);
	}


	{
		Profiler fl("file natural sort");
		while (!Devide(A, B, C))
		{

			Merge(A, B, C);

		}
	}
	sort_type* Fres = new sort_type[SZ];
	A.clear();
	A.seekg(0);
	A.read((char*)Fres, SZ * sizeof(sort_type));

	cout << "\n\is_equal: " << std::boolalpha << std::equal(cmp, cmp + SZ,Fres, Fres +
SZ);

	return 0;
	}
```

3.2.2 Програмна реалізація модифікованого алгоритму

General.h

```
#pragma once
#include <time.h>
#include <string>

typedef int sort_type;

constexpr size_t bytes_in_KB = 1024;
constexpr size_t bytes_in_MB = 1024 * 1024;
constexpr size_t pref_size = bytes_in_MB * 1;
```

General.cpp

```
#include "General.h"

#include <iostream>
using std::cerr;


Profiler::Profiler(std::string msg) : message(move(msg))
{
```

```
	start = clock();
}
Profiler::~Profiler()
{
	cerr << "\nEnd " << message << " duration: " << (clock() - start) << " msec" <<
std::endl;
}
```

Algs.cpp

```
#include "Algs.h"
#include <stdlib.h>
#include <time.h>
#include <random>
#include <execution>

#include <fstream>

void Generate(string filename, size_t file_size, int* raw_memory, size_t mem_size)
{
	FILE* file;
	fopen_s(&file, filename.c_str(), "wb");
	size_t num_of_el, been_write = 0, fit = file_size / sizeof(int);
	std::mt19937 Generator;

	int MN = time(0);

	Generator.seed(MN);
	int fit_size = fit * sizeof(int);
	do
	{
		num_of_el = std::min(mem_size / sizeof(int), fit - been_write);
		been_write += num_of_el;
		for (size_t i = 0; i < num_of_el; ++i)
		{
			raw_memory[i] = Generator();
		}
		fwrite(raw_memory, num_of_el * sizeof(int), 1, file);
	} while (fit != been_write);
	fclose(file);
}

bool Devide(MappedFile& A, MappedFile& B, MappedFile& C)
{
	A.ToBegin();
	B.Trunc();
	C.Trunc();

	bool flag = true;

	sort_type buf = 0, old = 0;


	old = A.Read();
	while (true)
	{
		MappedFile& Active = flag ? B : C;
		flag = !flag;

		do
		{
			Active.Write(old);

			buf = A.Read();
			if (A.Eof())
				break;
```

```
                        if (old <= buf)
                        {
                                old = buf;
                        }
                        else
                        {
                                old = buf;
                                break;
                        }

                } while (true);
                if (A.Eof())
                {
                        B.Flush();
                        C.Flush();
                        return C.Empty();
                }
        }
}


void Merge(MappedFile& A, MappedFile& B, MappedFile& C)
{
        A.ToBegin(false);
        B.ToBegin();
        C.ToBegin();


        sort_type old_B, old_C;
        old_B = B.Read();
        old_C = C.Read();
        sort_type buf_B, buf_C;

        while (true)
        {
                if (old_B <= old_C)
                {
                        A.Write(old_B);

                        buf_B = B.Read();
                        if (B.Eof())
                        {
                                old_B = buf_B;
                                goto appendc;
                        }
                        if (buf_B < old_B)
                        {
                                do
                                {
                                        A.Write(old_C);
                                        buf_C = C.Read();

                                        if (C.Eof())
                                        {
                                                old_B = buf_B;
                                                goto appendb;
                                                break;
                                        }
                                        if (old_C <= buf_C)
                                        {
                                                old_C = buf_C;
                                        }
                                        else
                                        {
                                                old_C = buf_C;
```

```
                                        break;
                                }
                        } while (true);
                }
                old_B = buf_B;
        }
        else
        {
                A.Write(old_C);

                buf_C = C.Read();
                if (C.Eof())
                {
                        old_C = buf_C;
                        goto appendb;
                }
                if (buf_C < old_C)
                {
                        do
                        {
                                A.Write(old_B);
                                buf_B = B.Read();
                                if (B.Eof())
                                {
                                        old_C = buf_C;
                                        goto appendc;
                                        break;
                                }
                                if (old_B <= buf_B)
                                {
                                        old_B = buf_B;
                                }
                                else
                                {
                                        old_B = buf_B;
                                        break;
                                }
                        } while (true);
                }
                old_C = buf_C;
        }
}
if (B.Eof())
{
appendc:
        A.Write(old_C);
        if (C.Eof())
        {
                A.Flush();
        }
        else
        {
                A += C;
        }
        return;
}
if (C.Eof())
{
appendb:
        A.Write(old_B);

        if (B.Eof())
        {
                A.Flush();
        }
```

```
                else
                {
                        A += B;
                }
                return;
        }

}


bool IsSorted(string filename, sort_type* raw_memory, size_t mem_size)
{
        FILE* file;
        fopen_s(&file,filename.c_str(), "rb");
        fseek(file, 0, SEEK_SET);

        sort_type left;
        sort_type right;

        size_t beg = 0;
        size_t end = fread(raw_memory, 1, mem_size, file);
        if (!std::is_sorted(raw_memory, raw_memory + end / sizeof(sort_type)))
        {
                fclose(file);
                return false;
        }
        while (true)
        {
                left = raw_memory[end / sizeof(sort_type) - 1];
                size_t beg = 0;
                size_t end = fread(raw_memory, 1, mem_size, file);
                right = raw_memory[0];
                if (feof(file))
                        break;
                if (!std::is_sorted(raw_memory, raw_memory + end / sizeof(sort_type)))
                {
                        fclose(file);
                        return false;
                }
                if(left > right)
                {
                        fclose(file);
                        return false;
                }
        }
        fclose(file);



        return true;
}

void PreSort(string filename, sort_type* raw_memory, size_t mem_size)
{
        std::fstream file(filename, std::ios_base::binary | std::ios_base::in |
std::ios_base::out);

        size_t end =0, old_pos = 0 ,
                been_read = 0;
        do
        {
                old_pos = file.tellg();
                file.read((char*)raw_memory, mem_size);
                end = file.gcount();
                std::sort(std::execution::par_unseq,
```

```
                raw_memory, raw_memory + end / sizeof(sort_type));
            file.seekp(old_pos, std::ios_base::beg);
            old_pos += end;
            DropFilePatially(file, (char*)raw_memory, end);
            file.flush();
        } while (!file.eof());

        file.clear();

        file.seekp(mem_size / 2, std::ios_base::beg);

        do
        {
            old_pos = file.tellg();
            file.read((char*)raw_memory, mem_size);
            end = file.gcount();
            std::sort(std::execution::par_unseq,
                raw_memory, raw_memory + end / sizeof(sort_type));
            file.seekp(old_pos, std::ios_base::beg);
            old_pos += end;
            DropFilePatially(file, (char*)raw_memory, end);
            file.flush();
        } while (!file.eof());

        file.clear();
        file.close();
}


void DropFilePatially(std::fstream& file, char* data, size_t size)
{
        size_t num = size / pref_size;
        for (size_t i = 0; i < num; i++)
        {
            file.write(data, pref_size);
            file.flush();
            data += pref_size;
        }
        file.write(data + pref_size * num, size - pref_size * num);
}
```

MappedFile.cpp

```
#include "MappedFile.h"

#include <filesystem>
#include <iostream>
#include <io.h>
#include <assert.h>
using std::ofstream;
using std::ios_base;


void MappedFile::read_chunk()
{
        assert(filesize != 0);
        to_read = std::min(chunk_size, filesize - been_read);
        fread(_data, to_read, 1, raw_file);
        been_read += to_read;

}

void MappedFile::write_chunk()
{
        to_write = std::min(chunk_size, pos * sizeof(sort_type));
```

```
	fwrite(_data, to_write, 1, raw_file);
	pos = 0;
}

MappedFile::MappedFile(string filename, sort_type* data, size_t size) : _filename(filename),
_data(data), chunk_size(size)
{
	FILE* dummy;
	fopen_s(&dummy, _filename.c_str(), "ab");
	fclose(dummy);

	fopen_s(&raw_file, _filename.c_str(), "rb+");
	setvbuf(raw_file, NULL, _IONBF, 0);

	filesize = std::filesystem::file_size(_filename);

	if (filesize) read_chunk();
}

MappedFile::~MappedFile()
{
	Close();
}

sort_type MappedFile::Read()
{
	if (pos * sizeof(sort_type) >= chunk_size)
	{
		pos = 0;
		read_chunk();
	}

	return _data[pos++];
}



void MappedFile::Write(sort_type buf)
{
	if (pos * sizeof(sort_type) >= chunk_size)
	{
		write_chunk();
	}
	_data[pos++] = buf;
}

bool MappedFile::Eof()
{
	return (pos * sizeof(sort_type) > to_read) || !to_read;
}

void MappedFile::ToBegin(bool read)
{
	rewind(raw_file);
	pos = been_read = 0;
	if (read) read_chunk();
	clearerr(raw_file);
}

void MappedFile::Trunc()
{
	_chsize(_fileno(raw_file), 0);

	filesize = 0;
```

```
	ToBegin(false);
}

bool MappedFile::Empty()
{

	return (std::filesystem::file_size(_filename) == 0);
}

void MappedFile::Flush()
{
	if (!been_read && pos)
	{
		write_chunk();
	}
	filesize = std::filesystem::file_size(_filename);;
	clearerr(raw_file);
	pos = 0;

}

void MappedFile::Close()
{
	fclose(raw_file);
}

void MappedFile::operator+=(MappedFile& Another)
{
	fwrite(this->_data, pos * sizeof(sort_type), 1, this->raw_file);
	fwrite(Another._data + Another.pos, Another.to_read - Another.pos *
sizeof(sort_type), 1, this->raw_file);

	if (Another.pos != 0)
	{
		while (Another.to_read == Another.chunk_size)
		{
			Another.read_chunk();
			fwrite(Another._data, Another.to_read, 1, this->raw_file);
		}
	}
	clearerr(this->raw_file);
	this->filesize = std::filesystem::file_size(this->_filename);
	this->pos = 0;
}
```

Source.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <time.h>
#include <random>

#include "Algs.h"

#include <Windows.h>

using std::cout;
using std::cin;

constexpr char fileAname[] = "A.bin";
constexpr char fileBname[] = "B.bin";
constexpr char fileCname[] = "C.bin";

constexpr size_t min_entities_for_sort = 4;
```

```
void Set(size_t &file_size ,size_t &mem_avail)
{
	size_t mult = 0;
	do
	{
		char ch;
		std::cout << "Choose Units: \n"
			"\t1 - bytes\n"
			"\t2 - KBs\n"
			"\t3 - MBs\n";
		cin >> ch;
		switch (ch)
		{
		case '2': mult = bytes_in_KB; break;
		case '3': mult = bytes_in_MB; break;
		case '1': mult = 1;

		}
	} while (!mult);

	size_t real_mem_avail, real_file_size;
	cout << "Enter file size: "; cin >> real_file_size;
	cout << "Eneter available memory: "; cin >> real_mem_avail;

	mem_avail = (real_mem_avail * mult / min_entities_for_sort / sizeof(sort_type)) *
sizeof(sort_type) * min_entities_for_sort;
	file_size = (real_file_size * mult / sizeof(sort_type) * sizeof(sort_type));
	cout << "fs: " << file_size << '\n';
	cout << "ma: " << mem_avail << '\n';

	if (mem_avail < sizeof(sort_type) * 8)
	{
		std::cerr << "UB: Too few memory!";
		exit(-1);
	}
	if (mem_avail >= file_size)
	{
		std::cerr << "Use internal sort!";
		exit(-1);
	}
}


int main()
{

	size_t mem_avail, file_size;
	Set(file_size, mem_avail);

	char* raw_data = new char[mem_avail];


	do
	{
		char ch;
		cout << "Generate | Sort | Check |Exit\n";
		cin >> ch;
		switch (ch)
		{
		case 'G':
		case 'g':
		{
			Profiler g("Generation");
			Generate(fileAname, file_size, (sort_type*)raw_data, mem_avail);
```

```
		}
		break;
		case 'S':
		case 's':
		{
			size_t peffered_mem_avail = mem_avail > pref_size ? pref_size :
mem_avail;
			{
				SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
				Profiler sort("Sorting " + std::to_string(file_size) + " bytes
with " + std::to_string(mem_avail) + " bytes memory available");

				{
					Profiler sort("Presort");
					PreSort(fileAname, (sort_type*)raw_data, mem_avail);
				}
				MappedFile A(fileAname, (sort_type*)raw_data, peffered_mem_avail
/ 2);
				MappedFile B(fileBname, (sort_type*)(raw_data +
peffered_mem_avail / 2), peffered_mem_avail / 4);
				MappedFile C(fileCname, (sort_type*)(raw_data + 3 *
peffered_mem_avail / 4), peffered_mem_avail / 4);

				while (!Devide(A, B, C))
				{
					Merge(A, B, C);
				}
				A.Close(); B.Close(); C.Close();
				SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS);
			}
			remove(fileBname);
			remove(fileCname);
			std::cout << "is sorted: " << std::boolalpha << IsSorted(fileAname,
(sort_type*)raw_data, mem_avail) << '\n';

		}
		break;
		case 'E':
		case 'e': return 0; break;
		case 'C':
		case 'c': std::cout << "is sorted: " << std::boolalpha << IsSorted(fileAname,
(sort_type*)raw_data, mem_avail) << '\n'; break;

		default:
			break;
		}

	} while (true);

	return 0;
}
```

ВИСНОВОК

При виконанні даної лабораторної роботи я познайомився з алгоритмами зовнішнього сортування. Мною був створений звичайний алгоритм, що відсортував 20 Мб за 16552 мілісекунд. Недоліком цього алгоритму є те, що він постійно звертається до пам’яті напряму.

Мною було створено обгортку над файлом, що використовує буфер для зберігання файлу також попередньо я пройшовся по файлу та відсортував його за допомогою внутрішнього сортування. Усі ці дії пришвидшили сортування 20 Мб з доступними 2 МБ пам’яті до 300 мілісекунд.

Обсяг ОП мого ПК – 16 ГБ. Я запустив мою програму для сортування фалу обсягом 32 ГБ та доступною пам’яттю 12 ГБ і отримав результат 3836296 мілісекунд, що становить майже 64 хвилини. Отже, на 1 ГБ у мене припадає ~ 2 хвилини.

Слід зазначити, що оскільки складність цього алгоритму є $O(n\ log(n)$, то зі збільшенням кількості байт, буде знижуватися швидкість сортування. Наприклад 32 МБ з 12 МБ доступними сортується зі швидкістю 21037 байт/с, а 32 ГБ з 12 ГБ доступними сортується зі швидкістю 8956 байт/с.

КРИТЕРІЇ ОЦІНЮВАННЯ

У випадку здачі лабораторної роботи до 09.10.2022 включно максимальний бал дорівнює – 5. Після 09.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- програмна реалізація алгоритму – 40%;
- програмна реалізація модифікацій – 40%;
- висновок – 5%.