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-15 Волинець Кирило Михайлович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Соколовський В.В.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 5

3.1 Псевдокод алгоритму 5

3.2 Програмна реалізація алгоритму 6

3.2.1 Вихідний код 6

ВИСНОВОК 24

КРИТЕРІЇ ОЦІНЮВАННЯ 25

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**
5	Пряме злиття

3 ВИКОНАННЯ

3.1 Псевдокод алгоритму

```
element_count = file.size()/NUMBER_SIZE
max_degree = get_max_degree(element_count)
sort_by_degree(file,1)
for degree = 2 to max_degree changing degree *= 2 do
      sort_by_degree(file,degree)
end for
sort_by_degree(file,max_degree)
```

3.1.1 max_degree(element_count):

```
return 2^(ceil(log2(element_count)) - 1)
```

3.1.2 sort_by_degree(file, degree):

```
divide_file(file,degree)
merge_by_degree(a_file,b_file,degree)
```

3.1.3 divide_file(input_file, degree):

```
element_count = input_file.size()/NUMBER_SIZE
degree_power = log2(degree)
for i = 0 to element_count do
      element = input_file.read()
      if get_bit(i,degree_power) == 1 do b_file.write(element)
      else do a_file.write(element)
end for
```

3.1.4 merge_by_degree(a_file, b_file, degree):

```
a_element_count = a_file.size()/NUMBER_SIZE
b_element_count = b_file.size()/NUMBER_SIZE

cycles_count = floor(b_element_count/degree)

a_element = a.read()
b_element = b.read()

for i to cycles_count do
      full_merge(degree,a_element,b_element,a_file,b_file,c_file)
end for

partial_merge(a_element,b_element,a_file,b_file,c_file)
```

3.1.5 full_merge (degree, a_element, b_element, a_file, b_file, c_file):

```
a_counter = 0
b_counter = 0
```

```
while a_counter<degree and b_counter<degree do
      if a_element<=b_element do
            c.write(a_element)
            a_element = a.read()
            a.counter += 1
      end if
      else do
            c.write(b_element)
            b_element = b.read()
            b.counter += 1
      end else
end while

if a_counter<degree do
      for a_counter to degree do
            c.write(a_element)
            a_element = a.read()
      end for
end if
elif b_counter<degree do
      for b_counter to degree do
            c.write(b_element)
            b_element = b.read()
      end for
end elif
```

3.1.6 partial_ merge(a_element, b_element, a_file, b_file, c_file):

```
while !a_file.eof() and !b_file.eof() do
      if a_element<=b_element do
            c.write(a_element)
            a_element = a.read()
      end if
      else do
            c.write(b_element)
            b_element = b.read()
      end else
end while

if !a_file.eof() do
      while !a_file.eof() do
            c.write(a_element)
            a_element = a.read()
      end while
end if
elif !b_file.eof() do
      while !b_file.eof() do
            c.write(b_element)
            b_element = b.read()
      end while
end elif
```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
#include <iostream>
#include "Header.h"
#include <random>
#include "ModSort.h"
```

```
#include "Sort.h"

using namespace std;

bool menu() {
    cout << "Choose action:\n";
    cout << "0 - generate file\n";
    cout << "1 - sort file\n";
    cout << "2 - check file sorting status\n";
    cout << "3 - transform file\n";

    int basic_choice;
    cin >> basic_choice;

    if (basic_choice == 0) {
        /*cout << "Enter type of file:\n";
        cout << "0 - text file\n";
        cout << "1 - binary file\n";*/

        int type_of_file = 1;
        //cin >> type_of_file;

        cout << "Enter file name:\n";
        string filename;
        cin >> filename;

        cout << "Enter max number (greater than 0 and less than 64bit
number limit):\n";
        unsigned long long max_n;
        cin >> max_n;

        if (type_of_file == 0) {
            cout << "Enter number of elements:\n";
            int num_of_elements;
            cin >> num_of_elements;

            generateFile(num_of_elements, filename, max_n);
        }
        else if (type_of_file == 1) {
            cout << "Enter size of file (in bytes, 1 element = 8
bytes):\n";
            int size;
            cin >> size;

            generateBinFile64(size, filename, max_n);
        }
        else return 0;
        cout << "File " << filename << " succesfully generated!\n";
    }
    else if (basic_choice == 1) {
        cout << "Enter type of sort:\n";
        cout << "0 - base (very slow)\n";
        cout << "1 - modified\n";
        cout << "2 - alternative (higher memory speed, less algorithm
speed)\n";
        int type_of_sort;
        cin >> type_of_sort;

        cout << "Enter file name (file must be binary):\n";
        string filename;
        cin >> filename;

        cout << "Enter output filename\n";
        string output_name;
```

```
            cin >> output_name;

            cout << "Delete intermediate products (0/1)?\n";
            bool delete_intermediate;
            cin >> delete_intermediate;

            if (type_of_sort == 0) {
                cout << "Show progress (0/1/2)?\n";
                int show_progress;
                cin >> show_progress;
                sortBin64File(filename, output_name, delete_intermediate,
show_progress);
            }
            else if (type_of_sort == 1) {
                cout << "Show progress (0/1)?\n";
                int show_progress;
                cin >> show_progress;

                cout << "Enter max amount of RAM (in bytes):\n";
                int max_RAM_size;
                cin >> max_RAM_size;

                sortBin64FileMod(filename, output_name, max_RAM_size,
delete_intermediate, show_progress);
            }
            else  if (type_of_sort == 2) {
                cout << "Show progress (0/1)?\n";
                int show_progress;
                cin >> show_progress;

                cout << "Enter max amount of RAM (in bytes):\n";
                int max_RAM_size;
                cin >> max_RAM_size;

                sortBin64FileAlt(filename, output_name, max_RAM_size,
delete_intermediate, show_progress);
            }
            else return 0;

            cout << "File " << filename << " succesfully sorted!\n";
        }
        else if (basic_choice == 2) {
            cout << "Enter file name:\n";
            string filename;
            cin >> filename;

            cout << "Enter file type:\n";
            cout << "0 - text\n";
            cout << "1 - bin64\n";
            int filetype;
            cin >> filetype;

            bool is_sorted;

            if (filetype == 0) {
                is_sorted = checkFileSortingText(filename);
            }
            else if (filetype == 1) {
                is_sorted = checkFileSortingBin64(filename);
            }
            else return 0;

            if(is_sorted) cout << "File " << filename << " is sorted\n";
            else cout << "File " << filename << " is NOT sorted\n";
```

```
        }
        else if (basic_choice == 3) {
            cout << "For now, only allowed bin64->text transforming\n";

            cout << "Enter file name (file must be binary):\n";
            string filename;
            cin >> filename;

            cout << "Enter output filename\n";
            string output_name;
            cin >> output_name;

            convertBin64toText(filename, output_name);

            cout << "File " << filename << " succesfully converted to
text!\n";
        }
        else return 0;
        cout << "\n";
        return 1;
    }

    int main()
    {
        while (menu());
    }
```

3.2.2 Header.h

```
    #pragma once
    #include <string>

    void generateFile(int count, std::string name, unsigned long long max =
ULLONG_MAX);
    void generateBinFile16(int size, std::string name, unsigned long long max
= USHRT_MAX);
    void generateBinFile32(int size, std::string name, unsigned long long max
= UINT16_MAX);
    void generateBinFile64(int size, std::string name, unsigned long long max
= ULLONG_MAX);

    void convertBin64toText(std::string bin64name, std::string outputname);

    bool checkFileSortingText(std::string filename);
    bool checkFileSortingBin64(std::string filename);
```

3.2.3 Functions.cpp

```
    #include "Header.h"
    #include "FileFunc.h"

    #include <random>
    #include <iostream>
    #include <fstream>

    typedef unsigned long long ull;

    void generateFile(int count, std::string name, ull max)
    {
        std::ofstream output(name);

        std::random_device seed;
```

```
	std::mt19937 engine(seed());

	std::uniform_int_distribution<size_t> generator(0, max);

	for (size_t i = 0; i < count; i++)
	{
		output << generator(engine) << " ";
	}

	output.close();
}
void generateBinFile16(int size, std::string name, ull max)
{
	std::ofstream output(name, std::fstream::binary);

	std::random_device seed;
	std::mt19937 engine(seed());

	std::uniform_int_distribution<unsigned short> generator(0, max);

	unsigned short n;

	if (max > USHRT_MAX) max = USHRT_MAX;

	int count = size / sizeof(unsigned short);

	for (size_t i = 0; i < count; i++)
	{
		n = generator(engine);
		output.write((char*)&n, sizeof(unsigned short));
	}

	output.close();
}
void generateBinFile32(int size, std::string name, ull max)
{
	std::ofstream output(name, std::fstream::binary);

	std::random_device seed;
	std::mt19937 engine(seed());

	std::uniform_int_distribution<unsigned int> generator(0, max);

	unsigned int n;

	if (max > UINT32_MAX) max = UINT32_MAX;

	int count = size / sizeof(unsigned int);

	for (size_t i = 0; i < count; i++)
	{
		n = generator(engine);
		output.write((char*)&n, sizeof(unsigned int));
	}

	output.close();
}
void generateBinFile64(int size, std::string name, ull max)
{
	std::ofstream output(name, std::fstream::binary);

	std::random_device seed;
	std::mt19937 engine(seed());
```

```
            std::uniform_int_distribution<ull> generator(0, max);

            ull n;

            int count = size / sizeof(ull);

            for (size_t i = 0; i < count; i++)
            {
                  n = generator(engine);
                  output.write((char*)&n, sizeof(ull));
            }

            output.close();
      }

      void convertBin64toText(std::string bin64name, std::string outputname) {
            std::fstream bin64file(bin64name, std::fstream::in |
std::fstream::binary);
            std::fstream textfile(outputname, std::fstream::out |
std::fstream::binary);

            int elementsCount = getFileSize(bin64name) / sizeof(ull);
            ull element;

            for (size_t i = 0; i < elementsCount; i++)
            {
                  bin64file.read((char*)&element, sizeof(ull));
                  textfile << element << " ";
            }

            bin64file.close();
            textfile.close();
      }

      bool checkFileSortingText(std::string filename)
      {
            ull a, b;
            std::ifstream file(filename);
            file >> a;
            while (file >> b) {
                  if (b < a) {
                        //std::cout << "a = " << a << ", b = " << b << "\n";
                        file.close();
                        return false;
                  }
                  a = b;
            }
            return true;
      }
      bool checkFileSortingBin64(std::string filename)
      {
            ull a, b;
            std::ifstream file(filename, std::ifstream::binary);

            int elementCount = getFileSize(filename) / sizeof(ull);
            file.read((char*)&a, sizeof(ull));

            for (size_t i = 1; i < elementCount; i++)
            {
                  file.read((char*)&b, sizeof(ull));
                  if (b < a) {
                        //std::cout << "a = " << a << ", b = " << b << "\n";
                        file.close();
                        return false;
```

```
                }
                a = b;
        }

        return true;
} include "stdafx.h"
```

3.2.4 FileFunc.h

```
#pragma once
#include <string>

int getFileSize(std::string filename);
```

3.2.5 FileFunc.cpp

```
#include "FileFunc.h"

int getFileSize(std::string filename) {
        struct stat buf;
        stat(filename.c_str(), &buf);
        return buf.st_size;
}
```

3.2.6 Sort.h

```
#pragma once
#include <string>

void sortBin64File(std::string fileName, std::string output, bool
deleteInter, int showProgress);
```

3.2.7 Sort.cpp

```
#include "Sort.h"
#include "FileFunc.h"

#include <fstream>
#include <cmath>
#include <iostream>

#include "Header.h"

typedef unsigned long long ull;

const std::string A_NAME("A"), B_NAME("B"), C_NAME("C");

int getMaxDegree(int elementCount) {
        float rawLog = log2(elementCount);
        return 1 << (short)(ceil(rawLog) - 1);
}

bool getBit(int n, int bit_pos) {
        return (n & (1 << bit_pos)) >> bit_pos;
}

int divideFile(std::string input_name, std::string a_name, std::string
b_name, int degree) {
        std::fstream input(input_name, std::fstream::in |
std::fstream::binary);
        std::fstream a(a_name, std::fstream::out | std::fstream::binary);
        std::fstream b(b_name, std::fstream::out | std::fstream::binary);
```

```
            int elementCount = getFileSize(input_name) / sizeof(ull);
            ull number;

            int degreePWR = log2(degree);

            for (size_t i = 0; i < elementCount; i++)
            {
                  input.read((char*)&number, sizeof(ull));
                  if (getBit(i, degreePWR)) b.write((char*)&number,
sizeof(ull));
                  else a.write((char*)&number, sizeof(ull));
            }

            input.close();
            a.close();
            b.close();

            return elementCount;
      }

      void fullMerge(std::fstream& a, std::fstream& b, std::fstream& c, int
degree, ull& a_element, ull& b_element) {

            int a_counter = 0;
            int b_counter = 0;

            while (a_counter < degree and b_counter < degree) {
                  if (a_element <= b_element) {
                        c.write((char*)&a_element, sizeof(ull));
                        a.read((char*)&a_element, sizeof(ull));
                        a_counter++;
                  }
                  else {
                        c.write((char*)&b_element, sizeof(ull));
                        b.read((char*)&b_element, sizeof(ull));
                        b_counter++;
                  }
            }
            if (a_counter < degree) {
                  for (a_counter; a_counter < degree; a_counter++)
                  {
                        c.write((char*)&a_element, sizeof(ull));
                        a.read((char*)&a_element, sizeof(ull));
                  }
            }
            else if (b_counter < degree) {
                  for (b_counter; b_counter < degree; b_counter++)
                  {
                        c.write((char*)&b_element, sizeof(ull));
                        b.read((char*)&b_element, sizeof(ull));
                  }
            }
      }

      void partialMerge(std::fstream& a, std::fstream& b, std::fstream& c, ull&
a_element, ull& b_element, int a_count, int b_count) {

            int a_counter = a.tellg() / sizeof(ull);
            int b_counter = b.tellg() / sizeof(ull);

            if (a_counter < 0) a_counter = a_count + 1;
            if (b_counter < 0) b_counter = b_count + 1;

            while (a_counter <= a_count and b_counter <= b_count) {
```

```
                if (a_element <= b_element) {
                        c.write((char*)&a_element, sizeof(ull));
                        a.read((char*)&a_element, sizeof(ull));
                        a_counter++;
                }
                else {
                        c.write((char*)&b_element, sizeof(ull));
                        b.read((char*)&b_element, sizeof(ull));
                        b_counter++;
                }
        }
        if (a_counter <= a_count) {
                for (a_counter; a_counter <= a_count; a_counter++)
                {
                        c.write((char*)&a_element, sizeof(ull));
                        a.read((char*)&a_element, sizeof(ull));
                }
        }
        else if (b_counter <= b_count) {
                for (b_counter; b_counter <= b_count; b_counter++)
                {
                        c.write((char*)&b_element, sizeof(ull));
                        b.read((char*)&b_element, sizeof(ull));
                }
        }
}

void mergeByDegree(std::string a_name, std::string b_name, std::string
c_name, int degree) {
        std::fstream a(a_name, std::fstream::in | std::fstream::binary);
        std::fstream b(b_name, std::fstream::in | std::fstream::binary);
        std::fstream c(c_name, std::fstream::out | std::fstream::binary);

        int a_count = getFileSize(a_name) / sizeof(ull);
        int b_count = getFileSize(b_name) / sizeof(ull);

        int full_b_cycles = b_count / degree;

        ull a_element;
        ull b_element;

        a.read((char*)&a_element, sizeof(ull));
        b.read((char*)&b_element, sizeof(ull));

        for (size_t i = 0; i < full_b_cycles; i++)
        {
                fullMerge(a, b, c, degree, a_element, b_element);
        }

        partialMerge(a, b, c, a_element, b_element, a_count, b_count);

        a.close();
        b.close();
        c.close();
}

void sortByDegree(std::string input_name, std::string a_name, std::string
b_name, std::string c_name, int degree, bool progress_showcase) {

        divideFile(input_name, a_name, b_name, degree);
        mergeByDegree(a_name, b_name, c_name, degree);

}
```

```
	void deleteABC() {
		remove((A_NAME + ".bin").c_str());
		remove((B_NAME + ".bin").c_str());
		remove((C_NAME + ".bin").c_str());
	}

	void sortBin64File(std::string filename, std::string output, bool
deleteInter, int progressShowcase) {
		std::fstream fileToSort(filename, std::fstream::in |
std::fstream::binary);
		int fileElementCount = getFileSize(filename)/sizeof(ull);
		int degree = 2;
		int maxDegree = getMaxDegree(fileElementCount);

		sortByDegree(filename, A_NAME + ".bin", B_NAME + ".bin", C_NAME +
".bin", 1, progressShowcase);

		if (progressShowcase >= 1) {
			std::cout << "Degree " << 1 << "done!\n";
		}
		if (progressShowcase == 2) {
			convertBin64toText("A.bin", "A1.txt");
			convertBin64toText("B.bin", "B1.txt");
			convertBin64toText("C.bin", "C1.txt");
		}


		for (degree; degree < maxDegree; degree *= 2)
		{
			sortByDegree(C_NAME + ".bin", A_NAME + ".bin", B_NAME +
".bin", C_NAME + ".bin", degree, progressShowcase);

			if (progressShowcase >= 1) {
				std::cout << "Degree " << degree << "done!\n";
			}
			if (progressShowcase == 2) {
				convertBin64toText("A.bin", "A" +
std::to_string((int)log2(degree) + 1) + ".txt");
				convertBin64toText("B.bin", "B" +
std::to_string((int)log2(degree) + 1) + ".txt");
				convertBin64toText("C.bin", "C" +
std::to_string((int)log2(degree) + 1) + ".txt");
			}
		}

		sortByDegree(C_NAME + ".bin", A_NAME + ".bin", B_NAME + ".bin",
output, degree, progressShowcase);

		if (progressShowcase >= 1) {
			std::cout << "Degree " << degree << "done!\n";
		}
		if (progressShowcase == 2) {
			convertBin64toText("A.bin", "A" +
std::to_string((int)log2(degree) + 1) + ".txt");
			convertBin64toText("B.bin", "B" +
std::to_string((int)log2(degree) + 1) + ".txt");
			convertBin64toText(output, "C" +
std::to_string((int)log2(degree) + 1) + ".txt");
		}

		if (deleteInter) deleteABC();

		fileToSort.close();
	}
```

3.2.8 ModSort.h

```
#pragma once
#include <string>

void sortBin64FileMod(std::string fileName, std::string output, int
allowedMemory, bool deleteInter, bool show_progress);
void sortBin64FileAlt(std::string filename, std::string output, int
allowedMemory, bool deleteInter, bool show_progress);
```

3.2.9 ModSort.cpp

```
#include "ModSort.h"
#include "FileFunc.h"

#include <cmath>
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

#include <sys/stat.h>

typedef unsigned long long ull;

const std::string NAME_DIVIDED_BASE = "D", NAME_SORTED_BASE = "S";

int getElementCount(std::fstream& file) {
      int counter = 0;
      ull element;
      file.seekg(0, std::ios_base::beg);
      while (file >> element) counter++;
      file.clear();
      file.seekg(0, std::ios_base::beg);
      return counter;
}

int chooseAdditionalFilesCount(int fileSize, int allowedMemory) {
      float rawCount = (float)fileSize / allowedMemory;
      float binLog = log2(rawCount);
      if (binLog < 0) binLog = 0;
      int exactCount = 1 << (int)ceil(binLog);
      return exactCount;
}
int chooseAdditionalFilesCountAlt(int fileSize, int allowedMemory) {
      float rawCount = (float)fileSize / allowedMemory;
      return ceil(rawCount);
}

void createDividesortBin64(std::fstream& file_to_read, int id, int seria,
int EPF) {
      std::fstream file_to_write;
      ull element;

      file_to_write.open(NAME_SORTED_BASE + std::to_string(seria) + "-" +
std::to_string(id) + ".bin", std::fstream::out | std::fstream::binary);

      ull* buffer = new ull[EPF];

      for (size_t i = 0; i < EPF; i++)
      {
```

```
                file_to_read.read((char*)&element, sizeof(ull));
                buffer[i] = element;
            }

            std::sort(&(buffer[0]), &(buffer[EPF]));


            for (size_t i = 0; i < EPF; i++)
            {
                file_to_write.write((char*)&(buffer[i]), sizeof(ull));
            }

            delete[] buffer;
            file_to_write.close();
        }
        void dividesortBin64(std::fstream& file, int fileCount, int fileSize) {
            int elementsCount = fileSize / sizeof(ull);
            double rawEPF = (double)elementsCount / fileCount;

            int effectiveEPF = ceil(rawEPF);
            int lastFileEPF = elementsCount - (fileCount - 1) * effectiveEPF;

            for (size_t i = 1; i < fileCount; i++)
            {
                createDividesortBin64(file, i, fileCount, effectiveEPF);
            }
            createDividesortBin64(file, fileCount, fileCount, lastFileEPF);
        }
        void internalSortBin64(std::fstream& file, std::string outputname, int
fileSize) {
            int elementsCount = fileSize / sizeof(ull);

            std::fstream outputFile;
            ull element;

            outputFile.open(outputname, std::fstream::out |
std::fstream::binary);

            ull* buffer = new ull[elementsCount];

            for (size_t i = 0; i < elementsCount; i++)
            {
                file.read((char*)&element, sizeof(ull));
                buffer[i] = element;
            }

            std::sort(&(buffer[0]), &(buffer[elementsCount]));


            for (size_t i = 0; i < elementsCount; i++)
            {
                outputFile.write((char*)&(buffer[i]), sizeof(ull));
            }

            delete[] buffer;
            outputFile.close();
        }

        void mergeFilesBin64(std::string file1name, std::string file2name,
std::string outputname) {
            std::fstream file1(file1name, std::fstream::in |
std::fstream::binary);
            std::fstream file2(file2name, std::fstream::in |
std::fstream::binary);
```

```
            std::fstream output(outputname, std::fstream::out |
std::fstream::binary);

            int file1size = getFileSize(file1name);
            int file2size = getFileSize(file2name);

            int file1count = file1size / sizeof(ull);
            int file2count = file2size / sizeof(ull);

            int file1counter = 0;
            int file2counter = 0;

            ull file1element;
            ull file2element;

            file1.read((char*)&file1element, sizeof(ull));
            file2.read((char*)&file2element, sizeof(ull));

            while (file1counter < file1count and file2counter < file2count) {
                  if (file1element >= file2element) {
                        output.write((char*)&file2element, sizeof(ull));
                        file2.read((char*)&file2element, sizeof(ull));
                        file2counter++;
                  }
                  else {
                        output.write((char*)&file1element, sizeof(ull));
                        file1.read((char*)&file1element, sizeof(ull));
                        file1counter++;
                  }
            }

            if (file1counter == file1count) {
                  for (file2counter; file2counter < file2count - 1;
file2counter++)
                  {
                        output.write((char*)&file2element, sizeof(ull));
                        file2.read((char*)&file2element, sizeof(ull));
                  }
                  output.write((char*)&file2element, sizeof(ull));
            }
            else if (file2counter == file2count) {
                  for (file1counter; file1counter < file1count - 1;
file1counter++)
                  {
                        output.write((char*)&file1element, sizeof(ull));
                        file1.read((char*)&file1element, sizeof(ull));
                  }
                  output.write((char*)&file1element, sizeof(ull));
            }

            file1.close();
            file2.close();
            output.close();
      }
      void mergePairsBin64(std::string nameBase, int fileCount, std::string
output) {
            std::string file1name, file2name, outputname;
            for (size_t i = 0; i < fileCount/2; i++)
            {
                  file1name = nameBase + std::to_string(fileCount) + "-" +
std::to_string(2*i + 1) + ".bin";
                  file2name = nameBase + std::to_string(fileCount) + "-" +
std::to_string(2*i + 2) + ".bin";
                  if (fileCount <= 2) outputname = output;
```

```
                  else outputname = nameBase + std::to_string(fileCount / 2) +
"-" + std::to_string(i + 1) + ".bin";
                  mergeFilesBin64(file1name, file2name, outputname);
            }
      }

      void deleteSeria(std::string nameBase, int fileCount) {
            std::string nameToDelete;
            for (size_t i = 1; i <= fileCount; i++)
            {
                  nameToDelete = nameBase + std::to_string(fileCount) + "-" +
std::to_string(i) + ".bin";
                  remove(nameToDelete.c_str());
            }

      }

      void sortBin64FileMod(std::string filename, std::string output, int
allowedMemory, bool deleteInter, bool show_progress)
      {
            std::fstream fileToSort(filename, std::fstream::in |
std::fstream::binary);
            int fileSize = getFileSize(filename);
            int fileCount = chooseAdditionalFilesCount(fileSize, allowedMemory);

            if (fileCount == 1) internalSortBin64(fileToSort, output, fileSize);

            else {
                  dividesortBin64(fileToSort, fileCount, fileSize);
                  if (show_progress) std::cout << "Internal sort complete!\n";

                  for (fileCount; fileCount > 1; fileCount /= 2)
                  {
                        mergePairsBin64(NAME_SORTED_BASE, fileCount, output);

                        if (show_progress) std::cout << "FileCount " <<
fileCount<<" complete!\n";

                        if (deleteInter) deleteSeria(NAME_SORTED_BASE,
fileCount);
                  }
            }

            fileToSort.close();
      }

      void altMerge(std::string nameBase, int fileCount, std::string outputname)
{
            std::vector<std::fstream*> fileArray(fileCount);
            std::vector<ull> currentElementArray(fileCount);

            std::vector<int> fileCounterArray(fileCount, 0);
            std::vector<int> fileElementCountArray(fileCount);

            int readedFileCount = 0;

            std::string filename;


            for (size_t i = 0; i < fileCount; i++)
            {
                  filename = nameBase + std::to_string(fileCount) + "-" +
std::to_string(i + 1) + ".bin";
```

```
                  fileArray[i] = new std::fstream(filename, std::fstream::in |
std::fstream::binary);

                  fileElementCountArray[i] = getFileSize(filename) /
sizeof(ull);
                  fileArray[i]->read((char*)&(currentElementArray[i]),
sizeof(ull));
            }

            std::ofstream outputFile(outputname, std::fstream::out |
std::fstream::binary);

            int i;

            while (readedFileCount < fileCount) {
                  i = std::distance(currentElementArray.begin(),
std::min_element(currentElementArray.begin(), currentElementArray.end()));

                  outputFile.write((char*)&(currentElementArray[i]),
sizeof(ull));
                  fileArray[i]->read((char*)&(currentElementArray[i]),
sizeof(ull));
                  fileCounterArray[i]++;

                  if (fileCounterArray[i] >= fileElementCountArray[i]) {
                        delete fileArray[i];
                        fileArray.erase(fileArray.begin() + i);
                        currentElementArray.erase(currentElementArray.begin() +
i);
                        fileCounterArray.erase(fileCounterArray.begin() + i);

      fileElementCountArray.erase(fileElementCountArray.begin() + i);

                        readedFileCount++;
                  }
            }


      }

      void sortBin64FileAlt(std::string filename, std::string output, int
allowedMemory, bool deleteInter, bool show_progress)
      {
            std::fstream fileToSort(filename, std::fstream::in |
std::fstream::binary);
            int fileSize = getFileSize(filename);
            int fileCount = chooseAdditionalFilesCountAlt(fileSize,
allowedMemory);

            if (fileCount == 1) internalSortBin64(fileToSort, output, fileSize);

            else {
                  dividesortBin64(fileToSort, fileCount, fileSize);
                  if (show_progress) std::cout << "internal sort complete!\n";
                  altMerge(NAME_SORTED_BASE, fileCount, output);
                  if (deleteInter) deleteSeria(NAME_SORTED_BASE, fileCount);
            }

            fileToSort.close();
      }
```

4 ПОРІВНЯНННЯ

Для пришвидшення роботи були зроблені 2 модифікації: перша буде називатися **mod** а друга **alt**.

Суть роботи модифікації **mod**:

Задається максимально можливий для використання об'єм пам'яті **mem**. Файл ділится на мінімальну кількість 2^n файлів, кожний з яких не перевищує mem. Кожний з файлів сортуємо внутрішнім сортуванням. Потім об'єднуємо пари файлів таким же чином, що і у сортуванні merge, поки не залишиться тільки 1 файл.

Суть роботи модифікації **alt**:

Задається максимально можливий для використання об'єм пам'яті **mem**. Файл ділится на мінімальну кількість **n** файлів, кожний з яких не перевищує mem. Кожний з файлів сортуємо внутрішнім сортуванням. Створюємо масив розміром **n**, що містить останній зчитаний елемент файлу **n**. Знаходимо з цих елементів мінімальний і записуємо у результючий файл, після чого зчитуємо новий елемент відповідного файлу. Робимо поки не зчитані всі файли.

Перевага **mod** над **alt** полягає в тому що у нього дуже мала алгоритмічна складність і майже весь час його виконання упирається у зчитування и запис.

Перевага **alt** над **mod** в тому, що незалежно від вхідних даних і розміру **mem** завжди буде тільки 2 цикла зчитування записа (розділення + збирання).

Тому при швидкому накопичувачі та відносно великому значенню **mem** краще працює алгоритм **mod**, а при швидкому процессорі і малому значенню mem швидше виходить **alt**.

Протестуємо программу на файлі розміром 32MB та дозволеним розміром оперативної пам'яті 8MB.

Базовий алгоритм – 45.94c

Алгоритм mod – 5.59c

Алгоритм alt – 7.77c

Час роботии алгоритму mod при сортуванні 1ГБ з доступною пам'ятю 512МБ – 173.58с

ВИСНОВОК

При виконанні даної лабораторної роботи я навчився робити алгоритм сортування, що не залежить від об'єму внутрішньої пам'яті а також модифікував його для пришвидшення роботи, що дозволило використовувати швидку внутрішню пам'ять але зберегло можливість сортувати великі файли.

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%.