

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 студент ІІІ-15, Дацьо Іван Іванович

Перевірив Соколовський Владислав Володимирович

Київ 2022

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

```

Метод eof(string filename)
    //відкриваємо потік файлу для читання
    fstream f;
    f.open(filename, ios::in | ios::binary | ios::ate)
    //створюємо позчик на кінець файлу
    eof = !f.tellg()
    //закриваємо потік
    f.close()
    Повернути eof

Метод merge(filemanager &manager)
    //Число файлів на вході
    count = manager.get_in()

    //створюємо масив серій

    length_of_series = new int[count]

    //загальна сума серій

    sum_of_series = 2
    Поки sum_of_series > 1 то
        current_length = 0;
        Повторити для i від 0 до i < count

            //якщо тут є пусті серії

            Якщо manager.input[i].empty_series то

                length_of_series[i] = 0
                manager.input[i].empty_series = manager.input[i].empty_series - 1

            Інакше

                Якщо !manager.input[i].real_series то

                    //якщо не має реальних серій

                    manager.index_swap(i, 0);

                Якщо manager.input[i].empty_series то

                    // Шукаємо наявність порожніх серій вже в новому файлі

                    i = i - 1
                    продовжити
                    // Починаючи спочатку, ігноруючи наступні твердження

                length_of_series[i] = manager.read(i)
                manager.input[i].real_series = manager.input[i].real_series - 1
                buffer.insert(pair <int, int>(manager.read(i), i))

            current_length += length_of_series[i] - 1;

    manager.write(0, current_length);
    manager.output[0].real_series++;

    buf = buffer.begin();
    Поки !buffer.empty() то

```

```

        //перемістити на початок
        buf = buffer.begin();
        manager.write(0, buf->first);
        Якщо length_of_series[buf->second] - 1 то

            buffer.insert(pair <int, int>(manager.read(buf->second), buf-
>second))
        // Стираємо but із буферу
        buffer.erase(buf)

        /* Вираховуємо суму серій*/
        sum_of_series = 0;
        Повторити для i від до count

            sum_of_series += manager.input[i].real_series +
manager.input[i].empty_series;

            sum_of_series += manager.output[0].real_series +
manager.output[0].empty_series;

Метод first_distribution(string filename, filemanager &manager)
    fstream f;

    //Відкриваємо згенерований початковий файл

    f.open(filename, ios::in | ios::binary);

    //Беремо число файлів
    counter = manager.get_out();

    //стврюємо вектор Фібаначчі з числа файлів

    fibonacci fib_vector(counter);

    //Створюємо впорядковану серію Фібаначчі

    fib_vector.make_order_vector();
    fib_vector.make_dist_mass();

    // поточна позиція

    current = 0;

    // позиція наступного
    next = 0;
    current_file = 0;
    //довжина серії
    length = 0;
    //стартова позиція
    start_pos = manager.output[current_file].file_object.tellg();

    //збільшуємо позицію тому що ми лише розпочали

    manager.output[current_file].real_series=
manager.output[current_file].real_series + 1

    //Резервуємо місце для запису тривалості серії

    manager.write(current_file, 0)

```

```

Читаємо
f.read((char *)&next, sizeof(int))

// Різниця між поточним розподілом і розподілом Фібоначчі

diff = new int[counter]
diff[0] = 0
Повторити для i від 1 до counter

    diff[i] = 1                                     //[0 1 1 1 1 1 1 1.

flag = 0                                           //exit flag
Поки true то

    /* запис серії */
    Поки current <= next то

        // записуємо взятий елемент у файл

        manager.write(current_file, next)

        //збільшуємо розмір серії

        length++;

        // робимо наступний вже взятим

        current = next;
        Якщо !f.eof()

            f.read((char *)&next, sizeof(int))

        Інакше

            // виходимо із циклу повністю
            flag = !flag
            break;

        Якщо (flag == 1) break

        /* міняємо місцями файли */
        end_pos = manager.output[current_file].file_object.tellg();
//запам'ятовуємо ост позицію у файлі
        manager.output[current_file].file_object.seekg(start_pos);
        //Переміщуємось на початок файлу

        //Запишіть довжину (замість зарезервованого нуля)
        manager.write(current_file, length);

        //Повертаємось до збереженої позиції, попередньо кінцевої. положення
        manager.output[current_file].file_object.seekg(end_pos);
        /* Для розподілу Фібоначчі згідно з табл */

        Якщо (current_file < counter - 1) && ((diff[current_file] + 1) ==
diff[current_file + 1])) то

            current_file = current_file +

1
// Різниця між поточним розподілом і розподілом Фібоначчі
    diff[current_file] = diff[current_file] -1;

```

```

        manager.output[current_file].real_series =
manager.output[current_file].real_series+ 1;

Інакше

    Якщо !diff[counter - 1]

        fib_vector.make_order_vector();
        fib_vector.make_dist_mass();
        Повторити для i від 0 до counter

            diff[i] = fib_vector.mass[i] - manager.output[i].real_series;

        current_file = 0;
        diff[current_file] = diff[current_file] - 1;
        manager.output[current_file].real_series++;

    /* Економія місця для запису довжини проходу */

    //Ми робимо поточну позицію початковою
    start_pos = manager.output[current_file].file_object.tellg();

    //Зарезервувати простір для довжини серії

    manager.write(current_file, 0);
    length = 0;

    //Пишемо елемент з наступної серії
    manager.write(current_file, next);
    length = length+1;
    current = next

    //Якщо файл не закінчився, ми читаємо наступний елемент

    Якщо !f.eof()

        f.read((char *)&next, sizeof(int));

    Інакше

        break

    // Запам'ятовування поточної (останньої) позиції

    end_pos = manager.output[current_file].file_object.tellg();

    //Переходимо на позицію, розрізаємо. раніше - замінити 0 на довжину ряду
manager.output[current_file].file_object.seekg(start_pos);

    //Запишіть довжину

    manager.write(current_file, length);

    //Повернення до збереженої позиції

    manager.output[current_file].file_object.seekg(end_pos);
    Повторити для i від 0 до counter

    //Випишуємо кількість порожніх рядів з масиву різниць

```



```

manager.output[i].empty_series = diff[i];

f.close()

```

Метод show_binary_file_length(string filename)

Якщо eof(filename)

Виводимо: << "[empty]" << endl;

Інакше

```

fstream f
f.open(filename, ios::in | ios::binary)
temp
length
Поки !f.eof() то

    f.read((char *)&length, sizeof(int));
    Якщо length

        Виводимо: << "[len: " << length << "]" "
        Поки length то

            f.read((char *)&temp, sizeof(int))
            Виводимо << temp << " ";
            length = length-1;

```

f.close()

Повертаємо 0;

/ Визначення кінця ряду */*

Метод amount_of_series(string filename)

```

fstream f
f.open(filename, ios::in | ios::binary);
quantity = 1
current = 0
next = 0
position_series.push_back(f.tellg())
Виводимо "Current position: " << 1 + f.tellg() << " :: "
f.read((char *)&current, sizeof(int));
Виводимо << "Series №" << quantity << "starts with value : " << current <<
endl
f.read((char *)&next, sizeof(int));
Поки !f.eof() то

    /*Якщо все в порядку*/
    Якщо next > current то

        f.read((char *)&current, sizeof(int));

Інакше

    quantity = quantity+1;
    Виводимо << "Current position: " << f.tellg() / sizeof(int) << " :: ";

```

```

        position_series.push_back(f.tellg());
        Виводимо<< "Series №" << quantity << " starts with value: " << next <<
endl;
        f.read((char *)&current, sizeof(int));
    }
    Якщо current > next то

        //зчитуємо наступний елемент

        f.read((char *)&next, sizeof(int));

    Інакше

        quantity = quantity + 1

        Вивести "Current position: " << f.tellg() / sizeof(int) << " :: ";
        position_series.push_back(f.tellg());
        Вивести : "Series №" << quantity << "starts with value : " << current
<< endl;

        //зчитати з файлу
        f.read((char *)&next, sizeof(int));

//закрити потік файлу

f.close();
повернути quantity

```

Метод show_vector_series()

```

    Повторити для i від 0 до position_series.size()

        вивести position_series[i]

```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Source.cpp:

```

#include "Polyphase_merge.h"
#include <ctime>
#include <windows.h>
#include <fstream>
#include <algorithm>
#define input_name "input.dat"
#define output_name "output.dat"
#define debug_file "debug_info.txt"
const int amount_of_numbers = 3000000;
//number of values to generate
const int amount_of_files = 5;
//Number of files involved
const bool debug_mode = 0; //1 - with debug information, 0 - without it
using namespace std;

/* Checking for the Fibonacci class */

```

```

void fibonacci_class_check()
{
    fibonacci vector(5);
    vector.make_order_vector();
    vector.make_dist_mass();
    for (int i = 0; i < 10; i++)
    {
        vector.make_order_vector();
        vector.make_dist_mass();
        for (int i = 0; i < vector.order; i++)
        {
            cout << vector.mass[i] << " ";
        }
        cout << endl;
    }
}

/* Show all output files */
void show_output_files(polyphase &sorting)
{
    for (int i = 0; i < amount_of_files - 1; i++)
    {
        string file_name = "file_output_" + to_string(i) + ".dat";
        cout << file_name << ": ";
        sorting.show_binary_file_length(file_name);
    }
}

/* sort function */
void sort_function(filemanager &manager, polyphase &sorting, bool show_output,
clock_t &start_dist, clock_t &end_dist, clock_t &start_merge, clock_t
&end_merge)
{
    start_dist = clock();
    sorting.first_distribution(input_name, manager);
    end_dist = clock();
    if (show_output)
    {
        cout << "Files contain (first distribution): " << endl;
        show_output_files(sorting);
    }
    /* Working with empty series */
    int min = manager.output[0].empty_series;
    for (int i = 1; i < amount_of_files - 1; i++)
    {
        if (min > manager.output[i].empty_series)
        {
            min = manager.output[i].empty_series;
            //We take the
            minimum number of empty series
        }
    }
    if (min != 0)
        //If not zero, then we
        continue, otherwise we skip
    {
        for (int i = 0; i < amount_of_files - 1; i++)
        {
            manager.output[i].empty_series -= min;
        }
        manager.input[0].empty_series += min;
    }
    manager.fileswap();
    cout << endl << "Sorting... ";
    start_merge = clock();
}

```

```

        sorting.merge(manager);
        end_merge = clock();
        cout << "Completed" << endl;
        manager.fileswap();
    }

    /* Debugging Information */
    void view_debug_info(polyphase &sorting)
    {
        cout << "File " << input_name << " contains: " << endl;
        int quantity = sorting.show_binary_file_eof(input_name);
        cout << "Number of generated values:" << quantity << endl << endl;
        cout << "Series check: " << endl;
        quantity = sorting.amount_of_series(input_name);
        cout << "Number of series: " << quantity << endl << endl;
    }

    int main()
    {
        filemanager manager(1, amount_of_files - 1);
        polyphase sorting;
        clock_t start_generate, end_generate, start_dist, end_dist, start_merge,
        end_merge;
        start_generate = clock();
        sorting.generate(input_name, amount_of_numbers, 1000);
        //1000 - randomness generation module (upper bound)
        end_generate = clock();
        if (debug_mode)
        {
            view_debug_info(sorting);
            //Display debug info if mode == 1
        }
        sort_function(manager, sorting, debug_mode, start_dist, end_dist,
        start_merge, end_merge); //1 - debug info, 0 - no debug info

        /* Writing the sort result to a file*/
        fstream f;
        f.open(output_name, ios::out | ios::binary);
        int length = manager.read(0);
        for (int i = 0; i < length; i++)
        {
            f << " " << manager.read(0);
        }
        //Write everything from input-a to a file
        f.close();
        cout << endl;

        /* Output on display */
        sorting.check_sort(output_name);
        float gen_t = ((double)end_generate - start_generate) /
        ((double)CLOCKS_PER_SEC);
        float dist_t = ((double)end_dist - start_dist) / ((double)CLOCKS_PER_SEC);
        float merge_t = ((double)end_merge - start_merge) /
        ((double)CLOCKS_PER_SEC);
        cout << "Amount of elements: " << amount_of_numbers << endl;
        cout << "Number of files: " << amount_of_files << endl;
        cout << "Generation time (generate): " << gen_t << " sec." << endl;
        cout << "Time of first distribution (first_dist): " << dist_t << " sec."
        << endl;
        cout << "sorting time (merge): " << merge_t << " sec." << endl << endl;
        return 0;
    }

```

Polyphase_merge.sort.h

```
#pragma once
#include <string>
#include <ctime>
#include <iostream>
#include <fstream>
#include <map>
#include "Filemanager.h"
#include "Fibonacci.h"

using namespace std;

/* Multiphase sort class */

class polyphase
{
public:
    vector<int> position_series; //A vector
    that stores the positions of the "breaks" of the series
    polyphase() {};
    void generate(string filename, int amount, int random_border);
    //Random number generator
    int show_binary_file_eof(string filename); //binary
    file output (until end of file)
    int show_binary_file_length(string filename); //Binary
    file output (based on run length)
    int amount_of_series(string filename); //Return
    number of series
    void show_vector_series(); //Displaying
    the positions of "breaks" of the series
    void show_txt_file(string filename); //BText
    file output
    int check_sort(string filename); //Sort check
    void first_distribution(string filename, filemanager &manager);
    //First distribution
    void merge(filemanager &manager); //Sort
    bool eof(string filename); //Checking for
    emptiness for a binary
};

/* First division, given that the file is binary.
*/

inline bool polyphase::eof(string filename)
{
    fstream f;
    f.open(filename, ios::in | ios::binary | ios::ate);
    bool eof = !f.tellg();
    f.close();
    return eof;
}

void polyphase::merge(filemanager &manager)
{
    multimap<int, int> buffer; //Value-
    ordered container: <value, file number>
```

```

    int count = manager.get_in(); //Number of
input files
    int *length_of_series = new int[count]; //array
of series
    int sum_of_series = 2; //Total sum of
series
    int current_length; //current
length

    while (sum_of_series > 1)
    {
        current_length = 0;
        for (int i = 0; i < count; i++)
        {
            if (manager.input[i].empty_series) //If there
are empty series
            {
                length_of_series[i] = 0;
                manager.input[i].empty_series--;
            }
            else
            {
                if (!manager.input[i].real_series) //If there
are no real series
                {
                    manager.index_swap(i, 0);
                    if (manager.input[i].empty_series) //We look
for the presence of empty series already in the new file
                    {
                        i--;
                        continue; //Starting over,
ignoring subsequent statements
                    }
                }
                length_of_series[i] = manager.read(i);
                manager.input[i].real_series--;
                buffer.insert(pair <int, int>(manager.read(i), i));
            }
            current_length += length_of_series[i]--;
        }

        manager.write(0, current_length);
        manager.output[0].real_series++;

        auto buf = buffer.begin();
        while (!buffer.empty())
        {
            buf = buffer.begin();
            manager.write(0, buf->first);
            if (length_of_series[buf->second]--)
            {
                buffer.insert(pair <int, int>(manager.read(buf->second), buf-
>second));
            }
            buffer.erase(buf);
        }

        /* Считаем сумму серий */
        sum_of_series = 0;
        for (int i = 0; i < count; i++)
        {
            sum_of_series += manager.input[i].real_series +
manager.input[i].empty_series;

```

```

    }
    sum_of_series += manager.output[0].real_series +
manager.output[0].empty_series;
}
}

void polyphase::first_distribution(string filename, filemanager &manager)
{
    fstream f; //Open the
generated input file
    f.open(filename, ios::in | ios::binary);
    int counter = manager.get_out(); //Grabbing
the number of output files
    fibonacci fib_vector(counter); //Create a
Fibonacci series of order counter
    fib_vector.make_order_vector(); //Create
the initial fibonacci series
    fib_vector.make_dist_mass();
    int current = 0; //Current
position
    int next = 0; //next to current
    int current_file = 0; //current file
    int length = 0; //Series
length
    streampos start_pos = manager.output[current_file].file_object.tellg();
//Start position (here = 0)
    streampos end_pos; //End position
    manager.output[current_file].real_series++; //We
increase the series by one (because we are starting)
    manager.write(current_file, 0); //We
reserve a place for recording the length of the series
    f.read((char *)&next, sizeof(int));

    int *diff = new int[counter]; //Difference
Between Current and Fibonacci Distribution
    diff[0] = 0;
    for (int i = 1; i < counter; i++)
    {
        diff[i] = 1; // [0 1 1 1 1 1 1
1 ...]
    }
    bool flag = 0; //exit flag
    while (true)
    {
        /* запись серии */
        while (current <= next)
        {
            manager.write(current_file, next); //We write
the taken element to the current file
            length++; //Increasing the
length of the series
            current = next; //We make the
current one already taken
            if (!f.eof())
            {
                //f >> next; //If the file is
not over, take the next one
                f.read((char *)&next, sizeof(int));
            }
            else
            {
                flag = !flag; //We exit this
and the outer loop completely

```

```

        break;
    }
}
if (flag == 1) break;

/* Меняем файлы */
end_pos = manager.output[current_file].file_object.tellg();
//Remember the current (last) position in the file
manager.output[current_file].file_object.seekg(start_pos);           //Move
to the beginning of the file
manager.write(current_file, length);                                 //Write
down the length (instead of the reserved zero)
manager.output[current_file].file_object.seekg(end_pos);             //We
return to the saved position previously finite. position

/* For fibonacci distribution according to the table*/

if ((current_file < counter - 1) && ((diff[current_file] + 1) ==
diff[current_file + 1]))
{
    current_file++;
//Difference Between Current and Fibonacci Distribution
    diff[current_file]--;
    manager.output[current_file].real_series++;
}
else
{
    if (!diff[counter - 1])
    {
        fib_vector.make_order_vector();
        fib_vector.make_dist_mass();
        for (int i = 0; i < counter; i++)
        {
            diff[i] = fib_vector.mass[i] - manager.output[i].real_series;
        }
    }
    current_file = 0;
    diff[current_file]--;
    manager.output[current_file].real_series++;
}

/* Saving space for recording the length of the run */
start_pos = manager.output[current_file].file_object.tellg();       //We
make the current position the starting position
manager.write(current_file, 0);                                     //Reserve
space for series length
length = 0;                                                         //Resetting
the series length

manager.write(current_file, next);                                   //We write
an element from the next series
length++;
current = next;
if (!f.eof())                                                         //If the file has
not ended, we read the next element
{
    f.read((char *)&next, sizeof(int));
}
else
{
    break;
}

```



```

    }
    end_pos = manager.output[current_file].file_object.tellg();
    //Remembering the current (last) position
    manager.output[current_file].file_object.seekg(start_pos);           //We move
    to the position, cut. earlier - replace 0 with the length of the series
    manager.write(current_file, length);                                   //Write down
    the length
    manager.output[current_file].file_object.seekg(end_pos);
    //Returning to the saved position
    for (int i = 0; i < counter; i++)
    {
        manager.output[i].empty_series = diff[i];                       //We write
        the number of empty series from the array of differences
    }
    f.close();
}

```

```

inline int polyphase::show_binary_file_length(string filename)

```

```

{
    if (eof(filename))
    {
        cout << "[empty]" << endl;
    }
    else
    {
        fstream f;
        f.open(filename, ios::in | ios::binary);
        int temp;
        int length;
        while (!f.eof())
        {
            f.read((char *)&length, sizeof(int));
            if (length)
            {
                cout << "[len: " << length << "]" << " ";
                while (length)
                {
                    int temp;
                    f.read((char *)&temp, sizeof(int));
                    cout << temp << " ";
                    length--;
                }
            }
            cout << endl;
            f.close();
        }
        return 0;
    }
}

```

```

inline void polyphase::generate(string filename, int amount, int random_border)

```

```

{
    fstream f;
    srand(time(NULL));
    f.open(filename, ios::out | ios::binary);
    for (int i = 0; i < amount - 1; i++)
    {
        int value = rand() % random_border;
        f.write((char*)&value, sizeof(int));
    }
    f.close();
}

```

```

inline int polyphase::check_sort(string filename)
{
    bool flag = 1;
    int counter = 1;
    int temp1, temp2;
    fstream f;
    f.open(filename, ios::in);
    f >> temp1;
    while (!f.eof())
    {
        f >> temp2;
        counter++;
        if (temp1 > temp2)
        {
            cout << "Sort error: " << counter << " in position ";
            flag = 0;
        }
        temp1 = temp2;
    }
    if (flag == true)
    {
        cout << "Sorting is correct: " << counter << " items have been sorted";
    }
    cout << endl;
    return flag;
}

inline int polyphase::show_binary_file_eof(string filename)
{
    fstream f;
    f.open(filename, ios::in | ios::binary);
    int temp;
    int quantity = 0;
    while (!f.eof())
    {
        int temp;
        f.read((char *)&temp, sizeof(int));
        cout << temp << " ";
        quantity++;
    }
    cout << endl;
    f.close();
    return quantity;
}

inline void polyphase::show_txt_file(string filename)
{
    fstream f;
    f.open(filename);
    int temp;
    while (!f.eof())
    {
        f >> temp;
        cout << temp << " ";
    }
    cout << endl;
    f.close();
}

/* Determining the end of a series */
int polyphase::amount_of_series(string filename)
{
    fstream f;

```

```

f.open(filename, ios::in | ios::binary);
int quantity = 1;
int current = 0;
int next = 0;
position_series.push_back(f.tellg());
cout << "Current position: " << 1 + f.tellg() << " :: ";
f.read((char *)&current, sizeof(int));
cout << "Series №" << quantity << "starts with value : " << current << endl;
f.read((char *)&next, sizeof(int));
while (!f.eof())
{
    /*if everything is OK*/
    if (next > current)
    {
        f.read((char *)&current, sizeof(int));
    }
    else
    {
        quantity++;
        cout << "Current position: " << f.tellg() / sizeof(int) << " :: ";
        position_series.push_back(f.tellg());
        cout << "Series №" << quantity << " starts with value: " << next <<
endl;
        f.read((char *)&current, sizeof(int));
    }
    if (current > next)
    {
        f.read((char *)&next, sizeof(int));
    }
    else
    {
        quantity++;
        cout << "Current position: " << f.tellg() / sizeof(int) << " :: ";
        position_series.push_back(f.tellg());
        cout << "Series №" << quantity << "starts with value : " << current <<
endl;
        f.read((char *)&next, sizeof(int));
    }
}
f.close();
return quantity;
}

inline void polyphase::show_vector_series()
{
    for (int i = 0; i < position_series.size(); i++)
    {
        cout << position_series[i] << " ";
    }
    cout << endl;
}

```

Filemanager.cpp

```
#include "Filemanager.h"
```

```
/* File manager class implementation */
```

```
filemanager::filemanager(int _in_count, int _out_count)
```

```

{
    in_count = _in_count;          /* take from function arguments (fixed bug) */
    out_count = _out_count;
}

```

```

    input.reserve(in_count);          /* reserve a place in vectors in advance */
    output.reserve(out_count);
    /* vs10 - long long, vs14+ - int */
    for (long long i = 0; i < in_count; i++)
    {
        string file_name = "file_input_" + to_string(i) + ".dat";
        input.push_back(file_definition(file_name));
        /* push fstream object and store file_name */
    }
    for (long long i = 0; i < out_count; i++)
    {
        string file_name = "file_output_" + to_string(i) + ".dat";
        output.push_back(file_definition(file_name));
        /* push fstream object and store file_name */
    }
}

void filemanager::fileswap()
{
    for (int i = 0; i < in_count; i++)
    {
        input[i].file_object.close();
        input[i].file_object.open(input[i].filename, ios::out | ios::binary);
    }
    for (int i = 0; i < out_count; i++)
    {
        output[i].file_object.close();
        output[i].file_object.open(output[i].filename, ios::in | ios::binary);
    }
    input.swap(output);
    //oh, hi - свапаем счетчики
    swap(in_count, out_count);
}

void filemanager::index_swap(int index_in, int index_out)
{
    input[index_in].file_object.close();
    input[index_in].file_object.open(input[index_in].filename, ios::out |
ios::binary);
    output[index_out].file_object.close();
    output[index_out].file_object.open(output[index_out].filename, ios::in |
ios::binary);
    input[index_in].swap(output[index_out]);
}

int filemanager::read(int index)
{
    int temp;
    input[index].file_object.read((char *)&temp, sizeof(int));
    return temp;
}

void filemanager::write(int index, int value)
{
    output[index].file_object.write((char*)&value, sizeof(int));          // output
array to file
}

bool filemanager::eof(int index)
{
    return input[index].file_object.eof();
}

```

```

int filemanager::get_in()
{
    return in_count;
}

int filemanager::get_out()
{
    return out_count;
}

filemanager::~filemanager()
{
    for (int i = 0; i < in_count; i++)
    {
        input[i].file_object.close();           //Close and delete unnecessary
files
        remove(input[i].filename.c_str());
    }
    for (int i = 0; i < out_count; i++)
    {
        output[i].file_object.close();
        remove(output[i].filename.c_str());
    }
}

/* We initialize the structure*/
filemanager::file_definition::file_definition(string _filename)
{
    filename = _filename;
    real_series = 0;
    empty_series = 0;
    file_object = fstream(filename, ios::out | ios::binary);
}

void filemanager::file_definition::swap(file_definition &rhs)
{
    file_object.swap(rhs.file_object);
    filename.swap(rhs.filename);
    std::swap(real_series, rhs.real_series);    //Coincidence of names is bad
    std::swap(empty_series, rhs.empty_series);
}

```

Filemanager.h

```

#pragma once
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

/*Filemanager for binary modification, working with files*/

class filemanager
{
private:
    struct file_definition           //Defining a file with a
struct
    {
        fstream file_object;        //The file-o-object itself
        string filename;            //Name
    }

```

```

        int real_series; //Number of real episodes
        int empty_series; //Number of empty series
        file_definition(string _filename); //Initialization
        void swap(file_definition &rhs);
    };
public:
    vector <file_definition> input, output; //We start the vector
of pairs "object - name"
    int in_count, out_count; //Counters for vectors
    filemanager(int in_count, int out_count); //Constructor
    void fileswap(); //Swap vectors
    void index_swap(int index_in, int index_out); //Swap specific vectors
by index
    int read(int index); //Read element from vector
    void write(int index, int value); //Write element to vector
    bool eof(int index); //End of file check
    ~filemanager(); //Destructor
    int get_in(); //return in counter
    int get_out(); //return out counter
};

```

Fibonccci.cpp

```

#include "Fibonacci.h"

/* Fibonacci class implementation    CPP */

/* Constructor */
fibonacci::fibonacci(int _order)
{
    current_pos = 0;
    order = _order; //take the input length
of the series
    index_end = order - 1; //determine the last
position in the vector
    for (int i = 0; i < order; i++)
    {
        fib_series.push_back(0); //all elements 0 except
one (=1)
    }
    fib_series[index_end] = 1; //last = 1
}
/* array that will be needed for
sorting itself (we sum the Fibonacci in the desired
okay) */
void fibonacci::make_dist_mass()
{
    mass = new int[order];
    int finish = current_pos + order;
    for (int i = 0; i < order; i++)
    {
        int temp = 0;
        for (int j = current_pos; j < finish; j++)
        {
            temp = temp + fib_series[j];
        }
        current_pos++;
        mass[i] = temp;
    }
    current_pos = finish - order + 1;
}
void fibonacci::make_order_vector()
{

```

```

    int temp1 = 0;
    int temp2 = 0;
    for (int i = 0; i < fib_series.size(); i++)
    {
        temp1 = temp1 + fib_series[i];
        if (i > order)
        {
            temp2 = temp2 + fib_series[i - order];
        }
        fib_series.push_back(temp1 - temp2);
    }
void fibonacci::show_fib_vector()
{
    for (int i = 0; i < fib_series.size(); i++)
    {
        cout << fib_series[i] << " ";
    }
}

void fibonacci::show_dist_mass()
{
    for (int i = 0; i < order; i++)
    {
        cout << mass[i] << " ";
    }
}

/*Compiling a fibonacci sequence in a vector
- we finish the numbers natural for the series into the vector */

```

Polyphase_merge.h

```

#pragma once
#include <string>
#include <ctime>
#include <iostream>
#include <fstream>
#include <map>
#include "Filemanager.h"
#include "Fibonacci.h"

using namespace std;

/* Multiphase sort class */

class polyphase
{
public:
    vector <int> position_series; //A vector
    that stores the positions of the "breaks" of the series
    polyphase() {};
    void generate(string filename, int amount, int random_border);
    //Random number generator
    int show_binary_file_eof(string filename); //binary
    file output (until end of file)
    int show_binary_file_length(string filename); //Binary
    file output (based on run length)
    int amount_of_series(string filename); //Return
    number of series

```

```

        void show_vector_series(); //Displaying
the positions of "breaks" of the series
        void show_txt_file(string filename); //BText
file output
        int check_sort(string filename); //Sort check
        void first_distribution(string filename, filemanager &manager);
//First distribution
        void merge(filemanager &manager); //Sort
        bool eof(string filename); //Checking for
emptiness for a binary
};

/* First division, given that the file is binary.
*/

inline bool polyphase::eof(string filename)
{
    fstream f;
    f.open(filename, ios::in | ios::binary | ios::ate);
    bool eof = !f.tellg();
    f.close();
    return eof;
}

void polyphase::merge(filemanager &manager)
{
    multimap <int, int> buffer; //Value-
ordered container: <value, file number>
    int count = manager.get_in(); //Number of
input files
    int *length_of_series = new int[count]; //array
of series
    int sum_of_series = 2; //Total sum of
series
    int current_length; //current
length

    while (sum_of_series > 1)
    {
        current_length = 0;
        for (int i = 0; i < count; i++)
        {
            if (manager.input[i].empty_series) //If there
are empty series
            {
                length_of_series[i] = 0;
                manager.input[i].empty_series--;
            }
            else
            {
                if (!manager.input[i].real_series) //If there
are no real series
                {
                    manager.index_swap(i, 0);
                    if (manager.input[i].empty_series) //We look
for the presence of empty series already in the new file
                    {
                        i--;
                        continue; //Starting over,
ignoring subsequent statements
                    }
                }
                length_of_series[i] = manager.read(i);
            }
        }
    }
}

```



```

        manager.input[i].real_series--;
        buffer.insert(pair <int, int>(manager.read(i), i));
    }
    current_length += length_of_series[i]--;
}

manager.write(0, current_length);
manager.output[0].real_series++;

auto buf = buffer.begin();
while (!buffer.empty())
{
    buf = buffer.begin();
    manager.write(0, buf->first);
    if (length_of_series[buf->second]--)
    {
        buffer.insert(pair <int, int>(manager.read(buf->second), buf->second));
    }
    buffer.erase(buf);
}

/* Считаем сумму серий */
sum_of_series = 0;
for (int i = 0; i < count; i++)
{
    sum_of_series += manager.input[i].real_series +
manager.input[i].empty_series;
}
    sum_of_series += manager.output[0].real_series +
manager.output[0].empty_series;
}
}

void polyphase::first_distribution(string filename, filemanager &manager)
{
    fstream f; //Open the
    generated input file
    f.open(filename, ios::in | ios::binary);
    int counter = manager.get_out(); //Grabbing
    the number of output files
    fibonacci fib_vector(counter); //Create a
    Fibonacci series of order counter
    fib_vector.make_order_vector(); //Create
    the initial fibonacci series
    fib_vector.make_dist_mass();
    int current = 0; //Current
    position
    int next = 0; //next to current
    int current_file = 0; //current file
    int length = 0; //Series
    length
    streampos start_pos = manager.output[current_file].file_object.tellg();
    //Start position (here = 0)
    streampos end_pos; //End position
    manager.output[current_file].real_series++; //We
    increase the series by one (because we are starting)
    manager.write(current_file, 0); //We
    reserve a place for recording the length of the series
    f.read((char *)&next, sizeof(int));

    int *diff = new int[counter]; //Difference
    Between Current and Fibonacci Distribution

```

```

diff[0] = 0;
for (int i = 1; i < counter; i++)
{
    diff[i] = 1;
1 ...]
}
bool flag = 0;
while (true)
{
    /* запись серии */
    while (current <= next)
    {
        manager.write(current_file, next);
        length++;
        current = next;
        if (!f.eof())
        {
            //f >> next;
            f.read((char *)&next, sizeof(int));
        }
        else
        {
            flag = !flag;
            break;
        }
    }
    if (flag == 1) break;

    /* Меняем файлы */
    end_pos = manager.output[current_file].file_object.tellg();
    manager.output[current_file].file_object.seekg(start_pos);
    manager.write(current_file, length);
    manager.output[current_file].file_object.seekg(end_pos);
    return to the saved position previously finite. position

    /* For fibonacci distribution according to the table*/

    if ((current_file < counter - 1) && ((diff[current_file] + 1) ==
diff[current_file + 1]))
    {
        current_file++;
        diff[current_file]--;
        manager.output[current_file].real_series++;
    }
    else
    {
        if (!diff[counter - 1])
        {
            fib_vector.make_order_vector();
            fib_vector.make_dist_mass();
            for (int i = 0; i < counter; i++)
            {
                diff[i] = fib_vector.mass[i] - manager.output[i].real_series;
            }
        }
    }
}

```

```

    }
}
current_file = 0;
diff[current_file]--;
manager.output[current_file].real_series++;
}

/* Saving space for recording the length of the run */
start_pos = manager.output[current_file].file_object.tellg(); //We
make the current position the starting position
manager.write(current_file, 0); //Reserve
space for series length
length = 0; //Resetting
the series length

manager.write(current_file, next); //We write
an element from the next series
length++;
current = next;
if (!f.eof()) //If the file has
not ended, we read the next element
{
    f.read((char *)&next, sizeof(int));
}
else
{
    break;
}
}
end_pos = manager.output[current_file].file_object.tellg();
//Remembering the current (last) position
manager.output[current_file].file_object.seekg(start_pos); //We move
to the position, cut. earlier - replace 0 with the length of the series
manager.write(current_file, length); //Write down
the length
manager.output[current_file].file_object.seekg(end_pos);
//Returning to the saved position
for (int i = 0; i < counter; i++)
{
    manager.output[i].empty_series = diff[i]; //We write
the number of empty series from the array of differences
}
f.close();
}

inline int polyphase::show_binary_file_length(string filename)
{
    if (eof(filename))
    {
        cout << "[empty]" << endl;
    }
    else
    {
        fstream f;
        f.open(filename, ios::in | ios::binary);
        int temp;
        int length;
        while (!f.eof())
        {
            f.read((char *)&length, sizeof(int));
            if (length)
            {
                cout << "[len: " << length << "]" << endl;
            }
        }
    }
}

```

```

        while (length)
        {
            int temp;
            f.read((char *)&temp, sizeof(int));
            cout << temp << " ";
            length--;
        }
    }
    cout << endl;
    f.close();
}
return 0;
}

inline void polyphase::generate(string filename, int amount, int random_border)
{
    fstream f;
    srand(time(NULL));
    f.open(filename, ios::out | ios::binary);
    for (int i = 0; i < amount - 1; i++)
    {
        int value = rand() % random_border;
        f.write((char*)&value, sizeof(int));
    }
    f.close();
}

inline int polyphase::check_sort(string filename)
{
    bool flag = 1;
    int counter = 1;
    int temp1, temp2;
    fstream f;
    f.open(filename, ios::in);
    f >> temp1;
    while (!f.eof())
    {
        f >> temp2;
        counter++;
        if (temp1 > temp2)
        {
            cout << "Sort error: " << counter << " in position ";
            flag = 0;
        }
        temp1 = temp2;
    }
    if (flag == true)
    {
        cout << "Sorting is correct: " << counter << " items have been sorted";
    }
    cout << endl;
    return flag;
}

inline int polyphase::show_binary_file_eof(string filename)
{
    fstream f;
    f.open(filename, ios::in | ios::binary);
    int temp;
    int quantity = 0;
    while (!f.eof())
    {

```

```

        int temp;
        f.read((char *)&temp, sizeof(int));
        cout << temp << " ";
        quantity++;
    }
    cout << endl;
    f.close();
    return quantity;
}

inline void polyphase::show_txt_file(string filename)
{
    fstream f;
    f.open(filename);
    int temp;
    while (!f.eof())
    {
        f >> temp;
        cout << temp << " ";
    }
    cout << endl;
    f.close();
}

/* Determining the end of a series */
int polyphase::amount_of_series(string filename)
{
    fstream f;
    f.open(filename, ios::in | ios::binary);
    int quantity = 1;
    int current = 0;
    int next = 0;
    position_series.push_back(f.tellg());
    cout << "Current position: " << 1 + f.tellg() << " :: ";
    f.read((char *)&current, sizeof(int));
    cout << "Series №" << quantity << "starts with value : " << current << endl;
    f.read((char *)&next, sizeof(int));
    while (!f.eof())
    {
        /*if everything is OK*/
        if (next > current)
        {
            f.read((char *)&current, sizeof(int));
        }
        else
        {
            quantity++;
            cout << "Current position: " << f.tellg() / sizeof(int) << " :: ";
            position_series.push_back(f.tellg());
            cout << "Series №" << quantity << " starts with value: " << next <<
endl;
            f.read((char *)&current, sizeof(int));
        }
        if (current > next)
        {
            f.read((char *)&next, sizeof(int));
        }
        else
        {
            quantity++;
            cout << "Current position: " << f.tellg() / sizeof(int) << " :: ";
            position_series.push_back(f.tellg());
            cout << "Series №" << quantity << "starts with value : " << current <<

```

```
endl;
    f.read((char *)&next, sizeof(int));
}
}
f.close();
return quantity;
}

inline void polyphase::show_vector_series()
{
    for (int i = 0; i < position_series.size(); i++)
    {
        cout << position_series[i] << " ";
    }
    cout << endl;
}
}
```

Modification :

```
#include "mod.h"
void mergeFiles(int &file_count) {
    //list of ifstream objects. since now all the ram_sized files need to be
open simultaneously
    vector<ifstream> input_files(file_count);

    //opening the files to read. file_name is being constructed on the fly by
using file_counh reference
    for(int i = 0; i < file_count; ++i) {
        string file_name = to_string(i) + ".txt";
        input_files[i].open(file_name);
    }
    //creating a file output.txr to write the sorted numbers
    ofstream output;
    output.open("output.dat", ios::out | ios::binary);

    //Node represents the node for each file. every time a element is read from
a file, ifstream updates the file position pointer to point to the next element
    int num;
    for(int i = 0; i < file_count; ++i) {
        //reading from file pointed by input_files[i]
        //int num;
        while (input_files[i] >> num){
            output.write((char*)&num, sizeof(int));
        }

    }
    //closing the opened files
    for(int i = 0; i < file_count; ++i) {
        string file_name = to_string(i) + ".txt";
        input_files[i].close();
    }
    output.close();
}

void createRAMFiles(string in_name, int ram_size, int &file_count) {
    ifstream inp_file;
    inp_file.open(in_name, ios::binary);
    ofstream out_file;
    vector<int> nums;
    bool more_input = true;
```

```

    int num_count = 0, num;
    string file_name = "";
    while(more_input) {
        //empty the array for reading contents for next file
        nums.clear();
        while(num_count < ram_size) {
            if(inp_file.read((char *)&num, sizeof(int))) { //reading number from
input file
                nums.push_back(num);
                num_count++;
            }
            else {
                //not able to read number. End of file reached
                more_input = false;
                break;
            }
        }
        if(nums.empty()) break;
        sort(nums.begin(), nums.end());
        file_name = to_string(file_count) + ".txt";

        //creates a file on demand and writes the sorted contents of nums
        out_file.open(file_name);
        for(int i = 0; i < nums.size(); ++i) {
            //writing to the file named 'file_name'
            out_file << nums[i] << " ";
        }
        //close the file. It's required because out_file object then can be used
to open a different file
        out_file.close();

        //re initializing for inserting ram_size elements again
        num_count = 0;

        // number of files created
        file_count++;
    }
    inp_file.close();
}

void preMod(string in_name , int ram_size ){
    int file_count = 0;
    createRAMFiles(in_name, ram_size, file_count);

    //Merge the files and write to a output.txt
    mergeFiles(file_count);
    remove(in_name.c_str());
    rename("output.dat" , "input.dat");
}

```

Демонстрація роботи

Робота програми:

```



Sorting... Completed

Sorting is correct: 3000000 items have been sorted
Amount of elements: 3000000
Number of files: 5
Generation time (generate): 0.16 sec.
Time of first distribution (first_dist): 9.575 sec.
sorting time (merge): 12.59 sec.

Process finished with exit code 0

```

Розміри файлів:

 input.dat	10.10.2022 1:29	DAT File	11 719 KB
 output.dat	10.10.2022 1:30	DAT File	11 394 KB




Модифікований алгоритм :

```

Sorting is correct: 3000000 items have been sorted
Amount of elements: 3000000
Number of files: 5
Generation time (generate): 0.15 sec.
Time of first distribution (first_dist): 0.984 sec.
sorting time (merge): 1.143 sec.

```

Розмір файлів :

 debug_info	09.10.2022 17:51	Text Document	8 KB
 input.dat	10.10.2022 1:35	DAT File	11 719 KB
 output.dat	10.10.2022 1:35	DAT File	11 395 KB

ВИСНОВОК

При виконанні даної лабораторної роботи я розробив та записав алгоритм зовнішнього сортування (Багатофазне сортування). Виконав програмну реалізацію алгоритму на мові програмування C++ та відсортував цим алгоритмом випадковим чином згенерований масив цілих чисел, що зберігається

у файлі .Також було модифіковано цей алгоритм , попередньо відсотуючи частинками (які поміщаються в оперативній пам'яті), Таким чином було досягнуто швидкості роботи алгоритму Багатофазного сортування , який вимагали в задачі .

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%.