**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 ІП-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.

1. 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 |

1. 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 кінць 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 Фібаначі

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; **Ящко** !f.eof()  
   
 f.read((**char** \*)&next, **sizeof**(**int**))  
   
 **Інакше**

**// виходимо із циклу повнісю**  
 flag = !flag **break**;  
   
   
 **Якщо** (flag == 1) **break**  
  
 */\* міняємо місцями файли \*/* end\_pos = manager.output[current\_file].file\_object.tellg(); *//запамятовуємо ост позицію у файлі* manager.output[current\_file].file\_object.seekg(start\_pos);

*//Переміщаємось на початок файлу*

*//* *Запишіть довжину (замість зарезервованого нуля)* manager.write(current\_file, length);

//Повертаємося 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**   
 **Якшо** !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

//Виписуємо 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;

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

//зактири 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 Sourse.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 - мrandomness 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); *//ВText 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*};

Fiboncci.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

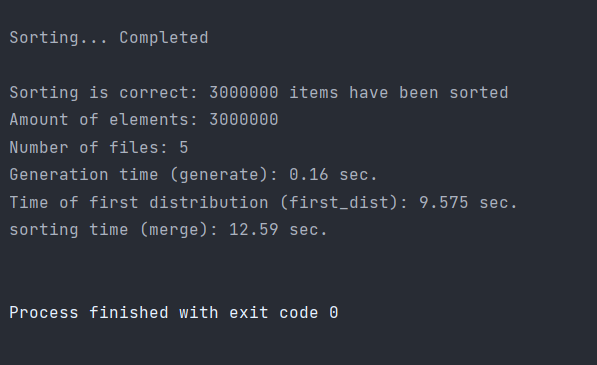
#pragma once  
#include **<string>**#include **<ctime>**#include **<iostream>**#include **<fstream>**#include **<map>**#include **"Filemanager.h"**#include **"Fibonacci.h"  
  
using namespace** std;  
  
*/\* Multiphase sort class \*/***class** polyphase  
{  
 **public**:  
 vector <**int**> position\_series; *//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); *//ВText 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;  
}

# 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

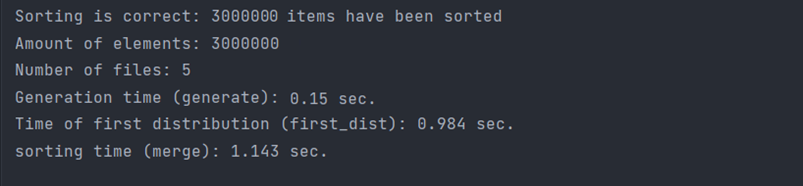
Робота програми:



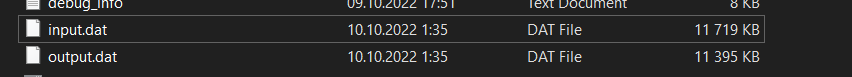
Розміри файлів:



Модифікований алгоритм :



Розмір файлів :



ВИСНОВОК

При виконанні даної лабораторної роботи я розробив та записав алгоритм зовнішнього сортування (Багатофазне сортування). Виконав програмну реалізацію алгоритму на мові програмування С++ 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%.