**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Проектування алгоритмів»

### „ Проектування і аналіз алгоритмів зовнішнього сортування”

**Виконав(ла)** ІП-25 Денисенко В.С.

(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Головченко М.М.*

(прізвище, ім'я, по батькові)

Київ 2023

ЗМІСТ

1. **МЕТА ЛАБОРАТОРНОЇ РОБОТИ .................................................... 3**
2. **ЗАВДАННЯ ............................................................................................ 4**
3. **ВИКОНАННЯ ........................................................................................ 6**
   1. ПСЕВДОКОД АЛГОРИТМУ ...................................................................... 6
   2. ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ .................................................... 6

#### 3.2.1Вихідний код .................................................................................. 6

**ВИСНОВОК .................................................................................................. 7**

**КРИТЕРІЇ ОЦІНЮВАННЯ ........................................................................ 8**

# МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

# ЗАВДАННЯ

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм зовнішнього сортування за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування та відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі (розмір файлу має бути не менше 10 Мб, можна значно більше).

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

Рекомендується попередньо впорядкувати серії елементів довжиною, що займає не менше 100Мб або використати інші підходи для пришвидшення процесу сортування.

Зробити узагальнений висновок з лабораторної роботи, у якому порівняти базову та модифіковану програми. У висновку деталізувати, які саме модифікації було виконано і який ефект вони дали.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм сортування** |
| 1 | Пряме злиття |
| 2 | Природне (адаптивне) злиття |
| 3 | Збалансоване багатошляхове злиття |
| 4 | Багатофазне сортування |
| 5 | Пряме злиття |
| 6 | Природне (адаптивне) злиття |
| 7 | Збалансоване багатошляхове злиття |
| 8 | Багатофазне сортування |
| 9 | Пряме злиття |
| 10 | Природне (адаптивне) злиття |
| 11 | Збалансоване багатошляхове злиття |
| 12 | Багатофазне сортування |
| 13 | Пряме злиття |
| 14 | Природне (адаптивне) злиття |
| 15 | Збалансоване багатошляхове злиття |
| 16 | Багатофазне сортування |
| 17 | Пряме злиття |
| 18 | Природне (адаптивне) злиття |
| 19 | Збалансоване багатошляхове злиття |
| 20 | Багатофазне сортування |
| 21 | Пряме злиття |
| 22 | Природне (адаптивне) злиття |
| 23 | Збалансоване багатошляхове злиття |
| 24 | Багатофазне сортування |
| 25 | Пряме злиття |
| 26 | Природне (адаптивне) злиття |
| 27 | Збалансоване багатошляхове злиття |
| 28 | Багатофазне сортування |
| 29 | Пряме злиття |
| 30 | Природне (адаптивне) злиття |
| 31 | Збалансоване багатошляхове злиття |
| 32 | Багатофазне сортування |
| 33 | Пряме злиття |
| 34 | Природне (адаптивне) злиття |
| 35 | Збалансоване багатошляхове злиття |

# ВИКОНАННЯ

3.1 Псевдокод алгоритму

void merge\_blocks(vector<int64\_t>& block1, vector<int64\_t>& block2) {

vector<int64\_t> merged\_block;

size\_t i = 0, j = 0;

// Злиття двох відсортованих блоків в один відсортований блок

while (i < block1.size() && j < block2.size()) {

if (block1[i] < block2[j]) {

merged\_block.push\_back(block1[i]);

++i;

} else {

merged\_block.push\_back(block2[j]);

++j;

}

}

// Додавання залишкових елементів

while (i < block1.size()) {

merged\_block.push\_back(block1[i]);

++i;

}

while (j < block2.size()) {

merged\_block.push\_back(block2[j]);

++j;

}

// Заміна вмісту блоків об'єднаним вмістом

block1 = merged\_block;

block2.clear();

}

void natural\_merge\_sort(vector<vector<int64\_t>>& blocks) {

while (blocks.size() > 1) {

for (size\_t i = 0; i < blocks.size() - 1; i += 2) {

// Злиття парних і непарних блоків

merge\_blocks(blocks[i], blocks[i + 1]);

}

// Якщо кількість блоків непарна, додаємо останній блок без змін

if (blocks.size() % 2 != 0) {

blocks[blocks.size() - 2] = blocks.back();

blocks.pop\_back();

}

}

}

3.2 Програмна реалізація алгоритму

## 3.2.1 Вихідний код

Main.cpp

#include"Sorting.h"

#include"ReadFile.h"

int main() {

fs::path input\_file = "C:\\Users\\slden\\Desktop\\input\_1000mb.bin";

fs::path output\_file = "C:\\Users\\slden\\Desktop\\output.bin";

fs::path file\_B = "C:\\Users\\slden\\Desktop\\file\_B.bin";

fs::path file\_C = "C:\\Users\\slden\\Desktop\\file\_C.bin";

auto start = std::chrono::high\_resolution\_clock::now();

SplitFile(input\_file, file\_B, file\_C);

while (true) {

int seq\_count = MargeFile(output\_file, file\_B, file\_C);

if (seq\_count== 1) {

break;

}

SplitFile(output\_file, file\_B, file\_C);

}

auto end = std::chrono::high\_resolution\_clock::now();

if (CheckSortedFile(output\_file)) {

std::cout << "File sorted!" << std::endl;

}

std::chrono::duration<float> duration = end - start;

std::cout << "Duration = " << duration.count() << " s " << std::endl;

return 0;

}

Sorting.h

#include <iostream>

#include <fstream>

#include <queue>

#include <vector>

#include<filesystem>

#include"ReadFile.h"

#include"WriteFile.h"

#include<chrono>

namespace fs = std::filesystem;

void SplitFile(fs::path input\_file, fs::path file\_B, fs::path file\_C);

void WriteSequenceToFile( ReadFile& src, WriteFile& dst, size\_t& offset);

int MargeFile(fs::path input\_file, fs::path file\_B, fs::path file\_C);

void MargeSequence(WriteFile& output, ReadFile& B\_buffer, ReadFile& C\_buffer, size\_t& B\_offset, size\_t& C\_offset);

void AddingBalanceToFile(ReadFile& src, WriteFile& dist, size\_t& offset);

bool CheckSortedFile(fs::path output\_file);

void PritBuffer(ReadFile& file);

Sorting.cpp

#include"Sorting.h"

void SplitFile(fs::path input\_file, fs::path file\_b, fs::path file\_C)

{

ReadFile input(input\_file);

WriteFile b\_buffer(file\_b), C\_buffer(file\_C);

size\_t element\_count = input.GetElementCount();

size\_t offset = 0;

int count = 0;

while (true) {

WriteSequenceToFile(input, b\_buffer, offset);

count++;

if (offset + 1 >= element\_count) {

break;

}

WriteSequenceToFile(input, C\_buffer, offset);

count++;

if (offset + 1 >= element\_count) {

break;

}

}

b\_buffer.Flush();

C\_buffer.Flush();

}

int MargeFile(fs::path output\_file, fs::path file\_b, fs::path file\_c)

{

size\_t b\_offset = 0, c\_offset = 0;

ReadFile b\_buffer(file\_b), c\_buffer(file\_c);

WriteFile output(output\_file);

size\_t b\_buffer\_size = b\_buffer.GetElementCount();

size\_t c\_buffer\_size = c\_buffer.GetElementCount();

int in1 = 0;

while (b\_buffer\_size > b\_offset && c\_buffer\_size > c\_offset) {

MargeSequence(output, b\_buffer, c\_buffer, b\_offset, c\_offset);

in1++;

//std::cout << in1 << ", " << b\_offset << ", " << c\_offset << std::endl;

}

output.Flush();

return in1;

}

void MargeSequence(WriteFile& output, ReadFile& b\_buffer, ReadFile& c\_buffer, size\_t& b\_offset, size\_t& c\_offset)

{

size\_t b\_buffer\_size = b\_buffer.GetElementCount();

size\_t c\_buffer\_size = c\_buffer.GetElementCount();

int64\_t number\_buffer\_b, number\_buffer\_c;

bool b\_seq\_end = true, c\_seq\_end = true;

if (b\_offset < b\_buffer\_size) {

number\_buffer\_b = b\_buffer.GetElement(b\_offset);

b\_seq\_end = false;

}

if (c\_offset < c\_buffer\_size) {

number\_buffer\_c = c\_buffer.GetElement(c\_offset);

c\_seq\_end = false;

}

while (!b\_seq\_end || !c\_seq\_end)

{

if ((number\_buffer\_b < number\_buffer\_c || c\_seq\_end) && !b\_seq\_end) {

output.AddElementToEnd(number\_buffer\_b);

b\_offset++;

bool is\_end\_of\_file = b\_offset == b\_buffer\_size;

if (is\_end\_of\_file) {

b\_seq\_end = true;

continue;

}

int64\_t next\_element = b\_buffer.GetElement(b\_offset);

b\_seq\_end = number\_buffer\_b > next\_element;

number\_buffer\_b = next\_element;

}

else if (!c\_seq\_end) {

output.AddElementToEnd(number\_buffer\_c);

c\_offset++;

bool is\_end\_of\_file = c\_offset == c\_buffer\_size;

if (is\_end\_of\_file) {

c\_seq\_end = true;

continue;

}

int64\_t next\_element = c\_buffer.GetElement(c\_offset);

c\_seq\_end = number\_buffer\_c > next\_element;

number\_buffer\_c = next\_element;

}

}

}

void Pritbuffer(ReadFile& file) {

size\_t size = file.GetElementCount();

for (size\_t i = 0; i < size; i++)

{

std::cout << file.GetElement(i) << std::endl;

}

std::cout << std::endl;

}

void AddingbalanceToFile(ReadFile& src, WriteFile& dist, size\_t& offset)

{

size\_t element\_count = src.GetElementCount();

int64\_t num = src.GetElement(offset);

while (true) {

dist.AddElementToEnd(num);

offset++;

if (offset >= element\_count) {

return;

}

num = src.GetElement(offset);

}

}

void WriteSequenceToFile(ReadFile& src, WriteFile& dst, size\_t& offset)

{

size\_t element\_count = src.GetElementCount();

if (element\_count == 0) {

return;

}

int64\_t num = src.GetElement(offset), next\_num;

dst.AddElementToEnd(num);

for (; offset < element\_count - 1; ) {

offset++;

next\_num = src.GetElement(offset);

if (num > next\_num) {

return;

}

num = next\_num;

dst.AddElementToEnd(num);

}

}

bool CheckSortedFile(fs::path output\_file)

{

ReadFile output(output\_file);

size\_t element\_count = output.GetElementCount();

int64\_t number = output.GetElement(0), next\_number;

for (size\_t i = 1; i < element\_count; i++)

{

next\_number = output.GetElement(i);

if (number > next\_number) {

return false;

}

number = next\_number;

}

return true;

}

ReadFile.h

#pragma once

#include<fstream>

#include<vector>

#include<filesystem>

namespace fs = std::filesystem;

class ReadFile

{

private:

static constexpr int buffer\_size = 1024 \*8;

int64\_t buffer[buffer\_size];

std::ifstream file;

int buffer\_offset;

void ReadFileToBuffer(int index);

bool IsElementInBuffer(int index);

public:

ReadFile(fs::path filepath);

int64\_t GetElement(int index);

size\_t GetElementCount();

};

ReadFile.cpp

#include"ReadFile.h"

ReadFile::ReadFile(fs::path filepath) {

file.open(filepath, std::ios::binary);

ReadFileToBuffer(0);

}

int64\_t ReadFile::GetElement(int index)

{

if (!IsElementInBuffer(index)) {

ReadFileToBuffer(index);

}

return buffer[index - buffer\_offset];

}

bool ReadFile::IsElementInBuffer(int index) {

return (index >= buffer\_offset) && (index <= buffer\_size + buffer\_offset - 1);

}

size\_t ReadFile::GetElementCount()

{

file.seekg(0, std::ios::end);

size\_t file\_size = file.tellg();

return file\_size / sizeof(int64\_t);

}

void ReadFile::ReadFileToBuffer(int index) {

index = std::max(0, index - 2);

int elements\_left\_to\_read = GetElementCount() - index;

int elements\_to\_read = std::min(buffer\_size, elements\_left\_to\_read);

file.seekg(index \* sizeof(int64\_t), std::ios\_base::beg);

file.read((char\*)&buffer, sizeof(int64\_t) \* elements\_to\_read);

buffer\_offset = index;

}

WriteFile.h

#pragma once

#include<fstream>

#include<vector>

#include<filesystem>

namespace fs = std::filesystem;

class WriteFile

{

private:

static constexpr int buffer\_size = 1024 \*8;

int64\_t buffer[buffer\_size];

int current\_position = 0;

std::ofstream file;

void WriteBufferToFile();

bool IsBufferFull();

public:

WriteFile(fs::path filepath);

void AddElementToEnd(int64\_t num);

void Flush();

};

WriteFile.cpp

#include"WriteFile.h"

WriteFile::WriteFile(fs::path filepath)

{

file.open(filepath, std::ios::binary);

}

void WriteFile::Flush()

{

WriteBufferToFile();

file.flush();

}

void WriteFile::AddElementToEnd(int64\_t num)

{

if (IsBufferFull()) {

WriteBufferToFile();

current\_position = 0;

}

buffer[current\_position] = num;

current\_position++;

}

void WriteFile::WriteBufferToFile() {

file.write((char\*)buffer, sizeof(int64\_t) \* current\_position);

}

bool WriteFile::IsBufferFull() {

return current\_position == buffer\_size - 1;

}

# ВИСНОВОК

При виконанні даної лабораторної роботи я реалізував алгоритм зовнішнього сортування, алгоритм природного (адаптивного) злиття. Головним завданням лабораторної було сортування масивів розміром значно більших за розмір використовуваної оперативної пам’яті, з чим мій алгоритм успішно справляється при сортування будь-яких розмірів масиву. Розрахунковий час вийшов за межі очікуваного, адже алгоритм сортує 1gb данних приблизно за 12хв, що все ще є хорошим показником, при використанні 800kb оперативної пам’яті.

При модифікуванні алгоритму, я змінив спосіб зчитання данних з файлу, тепер мій алгоритм зчитує до 1024byte данних, при будь-якому розмірі файлу. Дане покращення пришвидшило швидкодію алгоритму в декілька разів.

# КРИТЕРІЇ ОЦІНЮВАННЯ

У випадку здачі лабораторної роботи до 08.10.2022 включно максимальний бал дорівнює – 5. Після 08.10.2022 максимальний бал дорівнює –

4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 20%;
* програмна реалізація модифікацій – 20%;
* робота з git – 40%;  висновок – 5%.