**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**

Варіант 8

# Звіт

з лабораторної роботи № 1 з дисципліни
«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”

**Виконав(ла)** *ІП-13 Дойчев Костянтин* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

ЗМІСТ

## 1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

## 2. ЗАВДАННЯ

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

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

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше.

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

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

Таблиця 2.1 – Варіанти алгоритмів

| **№** | **Алгоритм сортування** |
|---|---|
| 1 | Пряме злиття |
| 2 | Природне (адаптивне) злиття |
| 3 | Збалансоване багатошляхове злиття |
| 4 | Багатофазне сортування |
| 5 | Пряме злиття |
| 6 | Природне (адаптивне) злиття |
| 7 | Збалансоване багатошляхове злиття |
| 8 | Багатофазне сортування |
| 9 | Пряме злиття |

| | |
|---|---|
| 10 | Природне (адаптивне) злиття |
| 11 | Збалансоване багатошляхове злиття |
| 12 | Багатофазне сортування |
| 13 | Пряме злиття |
| 14 | Природне (адаптивне) злиття |
| 15 | Збалансоване багатошляхове злиття |
| 16 | Багатофазне сортування |
| 17 | Пряме злиття |
| 18 | Природне (адаптивне) злиття |
| 19 | Збалансоване багатошляхове злиття |
| 20 | Багатофазне сортування |
| 21 | Пряме злиття |
| 22 | Природне (адаптивне) злиття |
| 23 | Збалансоване багатошляхове злиття |
| 24 | Багатофазне сортування |
| 25 | Пряме злиття |
| 26 | Природне (адаптивне) злиття |
| 27 | Збалансоване багатошляхове злиття |
| 28 | Багатофазне сортування |
| 29 | Пряме злиття |
| 30 | Природне (адаптивне) злиття |
| 31 | Збалансоване багатошляхове злиття |
| 32 | Багатофазне сортування |
| 33 | Пряме злиття |
| 34 | Природне (адаптивне) злиття |
| 35 | Збалансоване багатошляхове злиття |

# 3. ВИКОНАННЯ

## 3.1.Псевдокод алгоритму

```
function getClosestFibonacciSequence(number) {
    first = 0;
    second = 1;
    final = 1;
    while (final < number) {
        first = second;
        second = final;
        final = first + second;
    }

    return {
        first,
        second,
        final
    };
}

function getChunks(integers){
    chunks = [];
    tmp = [];
    for (let i = 0; i < integers.length - 1; i++) {
        isLast = i === integers.length - 2;
        tmp.push(integers[i]);
        if (integers[i] > integers[i + 1]) {
            chunks.push(tmp);
            tmp = [];
            if (isLast) {
                chunks.push([integers[i + 1]]);
            }
        }

        if (isLast && integers[i] <= integers[i + 1]) {
            tmp.push(integers[i + 1]);
            chunks.push(tmp);
        }
    }
    return chunks;
}


function sortExternally (initialFilePath, tmpDirPath) {
    integers := fs.readFile(initialFilePath);
    firstFilePath = path.join(tmpDirPath, 'first.bin');
    secondFilePath = path.join(tmpDirPath, 'second.bin');
    finalFilePath = path.join(tmpDirPath, 'final.bin');

    chunks = getChunks(integers);

    {first, second, final} = getClosestFibonacciSequence(chunks.length);
    firstFileContent = chunks.slice(0, first).flat();
    fs.writeFile(firstFilePath, firstFileContent);
    secondFileContent = chunks.slice(first).flat();
    fs.writeFile(secondFilePath, secondFileContent);

    sortedFilePath = ''

    function polyphaseMerge(firstFilePath, secondFilePath, finalFilePath)
{
        firstFileBuffer = fs.readFile(firstFilePath);
```

```
            secondFileBuffer = await fs.promises.readFile(secondFilePath);


            if (firstFileBuffer.length === 0 && secondFileBuffer.length === 0)
{
                return;
            }

             firstFileIntegers = Array.from(new
Int32Array(firstFileBuffer.buffer));
             secondFileIntegers = Array.from(new
Int32Array(secondFileBuffer.buffer));


             firstChunks: number[][] = getChunks(firstFileIntegers);
             secondChunks: number[][] = getChunks(secondFileIntegers);

             delta = firstChunks.length - secondChunks.length;

             finalChunks: number[][] = [];
            let iterableChunks: number[][] = firstChunks;
            if (delta > 0) {
                iterableChunks = secondChunks;
            }

            for (let i = 0; i < iterableChunks.length; i++) {
                 firstChunk = firstChunks[i];
                 secondChunk = secondChunks[i];
                finalChunks.push([...firstChunk, ...secondChunk].sort((a, b)
=> a - b));
            }

            let emptyFilePath = '';
            if (delta > 0) {
                 buffer = Buffer.from(new
Int32Array(firstChunks.slice(iterableChunks.length).flat()).buffer);
                await fs.promises.writeFile(firstFilePath, buffer, {encoding:
'binary', flag: 'w'});

                emptyFilePath = secondFilePath;
                await fs.promises.writeFile(secondFilePath, Buffer.from(new
Int32Array(0).buffer), {
                    encoding: 'binary',
                    flag: 'w'
                });
            } else if (delta < 0) {
                 buffer = Buffer.from(new
Int32Array(secondChunks.slice(iterableChunks.length).flat()).buffer);
                await fs.promises.writeFile(secondFilePath, buffer, {encoding:
'binary', flag: 'w'});

                emptyFilePath = firstFilePath;
                await fs.promises.writeFile(firstFilePath, Buffer.from(new
Int32Array(0).buffer), {
                    encoding: 'binary',
                    flag: 'w'
                });
            }


             finalFileContent = finalChunks.flat();
            await fs.promises.writeFile(finalFilePath,
Buffer.from(Int32Array.from(finalFileContent).buffer));
```

```
            if (delta === 0) {
                sortedFilePath = finalFilePath;
                return;
            }
             [first, second] = [firstFilePath, secondFilePath,
finalFilePath].filter((path) => path !== emptyFilePath);

            await polyphaseMerge(first, second, emptyFilePath);
        }

        await polyphaseMerge(firstFilePath, secondFilePath, finalFilePath);
    }
```

### 3.2.Програмна реалізація алгоритму

#### 3.2.1.Вихідний код

```
// main.ts

import path from "path";
import fs from "fs";

const file = path.join(__dirname, "../files/data.bin");
const output = path.join(__dirname, "../files/output.bin");

const getClosestFibonacciSequence = (number: number) => {
    let first = 0;
    let second = 1;
    let final = 1;
    while (final < number) {
        first = second;
        second = final;
        final = first + second;
    }

    return {
        first,
        second,
        final
    };
}

const getChunks = (integers: number[]) => {
    const chunks: number[][] = [];
    let tmp: number[] = [];
    for (let i = 0; i < integers.length - 1; i++) {
        const isLast = i === integers.length - 2;
        tmp.push(integers[i]);
        if (integers[i] > integers[i + 1]) {
            chunks.push(tmp);
            tmp = [];
            if (isLast) {
                chunks.push([integers[i + 1]]);
            }
        }

        if (isLast && integers[i] <= integers[i + 1]) {
            tmp.push(integers[i + 1]);
            chunks.push(tmp);
        }
```

```
        }
        return chunks;
    }

    const displayChunkOfFile = async (filePath: string, chunkSize: number) =>
{
        const readStream = fs.createReadStream(filePath, {highWaterMark:
chunkSize});
        let iteration = 0;
        readStream.on('data', (chunk) => {
            const data = Buffer.isBuffer(chunk) ? chunk : Buffer.from(chunk);
            const integers = Array.from(new Int32Array(data.buffer));
            if (iteration === 1) {
                readStream.destroy();
                return;
            }
            console.log(integers);
            iteration++;
        })

    }

    const sortExternally = async (initialFilePath: string, tmpDirPath: string)
=> {
        console.time("Sort time");
        const data = await fs.promises.readFile(initialFilePath);
        const integers = Array.from(new Int32Array(data.buffer));
        const firstFilePath = path.join(tmpDirPath, 'first.bin');
        const secondFilePath = path.join(tmpDirPath, 'second.bin');
        const finalFilePath = path.join(tmpDirPath, 'final.bin');

        const chunks = getChunks(integers);

        const {first, second, final} =
getClosestFibonacciSequence(chunks.length);
        const firstFileContent = chunks.slice(0, first).flat();
        await fs.promises.writeFile(firstFilePath,
Buffer.from(Int32Array.from(firstFileContent).buffer), {
            encoding: 'binary',
            flag: 'w'
        });
        const secondFileContent = chunks.slice(first).flat();
        await fs.promises.writeFile(secondFilePath,
Buffer.from(Int32Array.from(secondFileContent).buffer), {
            encoding: 'binary',
            flag: 'w'
        });

        let sortedFilePath = ''

        const polyphaseMerge = async (firstFilePath: string, secondFilePath:
string, finalFilePath: string) => {
            const firstFileBuffer = await fs.promises.readFile(firstFilePath);
            const secondFileBuffer = await
fs.promises.readFile(secondFilePath);


            if (firstFileBuffer.length === 0 && secondFileBuffer.length === 0)
{
                return;
            }

            const firstFileIntegers = Array.from(new
Int32Array(firstFileBuffer.buffer));
```

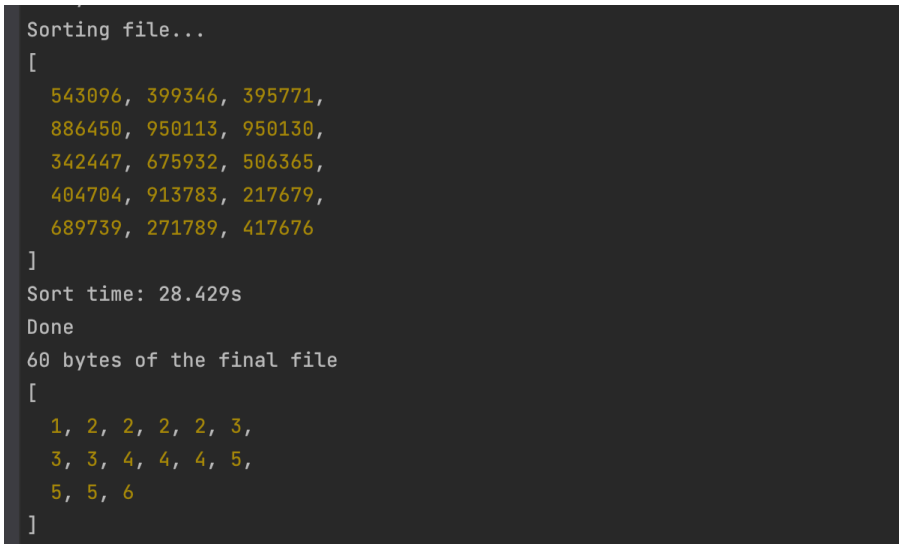
```
        const secondFileIntegers = Array.from(new
Int32Array(secondFileBuffer.buffer));


        const firstChunks: number[][] = getChunks(firstFileIntegers);
        const secondChunks: number[][] = getChunks(secondFileIntegers);

        const delta = firstChunks.length - secondChunks.length;

        const finalChunks: number[][] = [];
        let iterableChunks: number[][] = firstChunks;
        if (delta > 0) {
            iterableChunks = secondChunks;
        }

        for (let i = 0; i < iterableChunks.length; i++) {
            const firstChunk = firstChunks[i];
            const secondChunk = secondChunks[i];
            finalChunks.push([...firstChunk, ...secondChunk].sort((a, b)
=> a - b));
        }

        let emptyFilePath = '';
        if (delta > 0) {
            const buffer = Buffer.from(new
Int32Array(firstChunks.slice(iterableChunks.length).flat()).buffer);
            await fs.promises.writeFile(firstFilePath, buffer, {encoding:
'binary', flag: 'w'});

            emptyFilePath = secondFilePath;
            await fs.promises.writeFile(secondFilePath, Buffer.from(new
Int32Array(0).buffer), {
                encoding: 'binary',
                flag: 'w'
            });
        } else if (delta < 0) {
            const buffer = Buffer.from(new
Int32Array(secondChunks.slice(iterableChunks.length).flat()).buffer);
            await fs.promises.writeFile(secondFilePath, buffer, {encoding:
'binary', flag: 'w'});

            emptyFilePath = firstFilePath;
            await fs.promises.writeFile(firstFilePath, Buffer.from(new
Int32Array(0).buffer), {
                encoding: 'binary',
                flag: 'w'
            });
        }


        const finalFileContent = finalChunks.flat();
        await fs.promises.writeFile(finalFilePath,
Buffer.from(Int32Array.from(finalFileContent).buffer));

        if (delta === 0) {
            sortedFilePath = finalFilePath;
            return;
        }
        const [first, second] = [firstFilePath, secondFilePath,
finalFilePath].filter((path) => path !== emptyFilePath);

        await polyphaseMerge(first, second, emptyFilePath);
    }
```

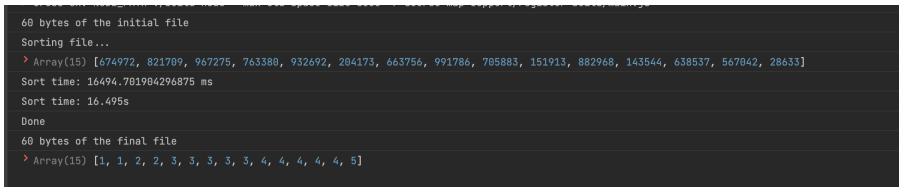
```
    await polyphaseMerge(firstFilePath, secondFilePath, finalFilePath);
    console.timeEnd("Sort time");

    await fs.promises.copyFile(sortedFilePath, output);
    await fs.promises.rm(tmpDirPath, {recursive: true, force: true});
}


export const main = async (filePath: string) => {
    const fileStats = await fs.promises.stat(filePath);
    const fileSize = fileStats.size;
    const tmp = await (async () => {
        try {
            await fs.promises.access(path.join(__dirname, '../files/
tmp'));
            return path.join(__dirname, '../files/tmp');
        } catch (_) {
            return await fs.promises.mkdir(path.join(__dirname, "../files/
tmp"), {recursive: true});
        }
    })();

    if (!tmp) {
        console.error("Failed to create tmp directory");
        process.exit(1);
    }

    console.log("60 bytes of the initial file");
    await displayChunkOfFile(filePath, 60);

    console.log("Sorting file...");
    await sortExternally(filePath, tmp);
    console.log("Done");

    console.log("60 bytes of the final file");
    await displayChunkOfFile(output, 60);
}

main(file);

// create-initial-file.ts

import fs from "fs";
import yargs from "yargs";
import {hideBin} from "yargs/helpers";

const MIN_FILE_SIZE_IN_BYTES = 10 * 1024 * 1024;
const MAX_BUFFER_SIZE_IN_BYTES = 20 * 1024 * 1024;
const INT_SIZE_IN_BYTES = 4;
const MAX_INT = Math.pow(2, 32) / 2 - 1;
const MIN_INT = -Math.pow(2, 32) / 2;

const argv = yargs(hideBin(process.argv))
    .usage("Usage: $0 --file [string] --size [number] --min [number] --max
[number]")
    .options({
        file: {type: "string", demandOption: true, alias: "f",
description: "File name"},
        size: {
            type: "number",
            alias: "s",
            default: MIN_FILE_SIZE_IN_BYTES,
            description: "File size in bytes",
            defaultDescription: "10MB"
```

```
            },
            min: {type: "number", default: 0, description: "Min integer"},
            max: {type: "number", default: 1000, description: "Max integer"},
        })
        .check(({size, min, max}) => {
            // if (size < MIN_FILE_SIZE_IN_BYTES) {
            //     throw new Error(`File size should be at least ${MIN_FILE_SIZE_IN_BYTES} bytes`);
            // }
            if (min > max) {
                throw new Error("Min should be less than max");
            }

            if (!Number.isInteger(min) || !Number.isInteger(max)) {
                throw new Error("Min and max should be integers");
            }

            if (max > MAX_INT) {
                throw new Error(`Max should be less or equal to the positive signed 32-bit integer. Max value is ${MAX_INT}`);
            }

            if (min < MIN_INT) {
                throw new Error(`Min should be greater of equal to the negative signed 32-bit integer. Min value is ${MIN_INT}`);
            }

            return true;
        })
        .help('h')
        .alias('h', 'help')
        .parseSync();

    const randomInteger = (min: number, max: number) => {
        return Math.floor(Math.random() * (max - min + 1)) + min;
    }

    const createFile = () => {
        const {file, max, min, size} = argv;
        const writeStream = fs.createWriteStream(file, {flags: "w", encoding: "binary"});
        if (size < MAX_BUFFER_SIZE_IN_BYTES) {
            const integers = new Array(Math.floor(size / INT_SIZE_IN_BYTES)).fill(0).map(() => randomInteger(min, max));
            const buffer = Buffer.from(Int32Array.from(integers).buffer);
            writeStream.write(buffer, "binary");
            writeStream.end();
            return;
        }

        let remainingBytes = size;
        const parts = Math.floor(size / MAX_BUFFER_SIZE_IN_BYTES);
        for (let i = 0; i < parts; i++) {
            const size = remainingBytes > MAX_BUFFER_SIZE_IN_BYTES ? MAX_BUFFER_SIZE_IN_BYTES : remainingBytes;
            const integers = new Array(Math.floor(size / INT_SIZE_IN_BYTES)).fill(0).map(() => randomInteger(min, max));
            const buffer = Buffer.from(Int32Array.from(integers).buffer);
            writeStream.write(buffer, "binary");
            remainingBytes -= size;
        }

        writeStream.end();
    }
```

```
createFile();
```

ВИСНОВОК




При виконанні даної лабораторної роботи ознайомився з алгоритмами зовнішнього сортування використовуючи polyphase merge sort.

Бінарний файл розміром в 10 МБ сортується приблизно за 22 секунд

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%.