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-13 Паламарчук Олександр Олександрович

Київ 2022

Зміст

Варіант 21

♦ 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.

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.

♦ Виконання

- Псевдокод алгоритму

  Основний алгоритм sort

  ЯКЩО (length) кількість чисел у вхідному файлі <= 1, ТО вивести повідомлення, що дані вже відсортовані.

  Присвоїти digitsNum = 1

  ПОКИ digitsNum (кількість цифр в групі) < length, ТО викликаємо метод distribution, викликаємо метод merge, збільшуємо digitsNum в два рази

  КІНЕЦЬ sort

  МЕТОД distribution

  ЦИКЛ проходу по всіх групах

  ЦИКЛ проходу по всім числам в групі

  ЯКЩО група має парний індекс, ТО читаємо число з вхідного файлу і записуємо його до першого буфера,

  ІНАКШЕ читаємо число з вхідного файлу і записуємо його до другого буфера

  КІНЕЦЬ ЦИКЛА

  КІНЕЦЬ ЦИКЛА

  КІНЕЦЬ distribution

  МЕТОД merge

Заповнити двовимірний масив (values) числами, де кількість рядків - це кількість буферів, а кількість колонок - це кількість чисел в групі.

Присвоїти pBuff1 = 0 (вказівник на групу першого буфера)

Присвоїти pBuff2 = 0 (вказівник на групу другого буфера)

ЦИКЛ проходу по всім елементам вхідного файлу

ЯКЩО values[BUFF_1 (індекс першого буфера)].length (довжина групи першого буфера) <= pBuff1 і values[BUFF_2 (індекс другого буфера)].length (довжина групи другого буфера) <= pBuff2, ТО заповнити values наступними групами з першого та другого буферів, і присвоїти pBuff1 = 0, pBuff2 = 0

Присвоїти indexOfMin = індекс буфера з мінімальним значенням на позиціях вказівників.

Записуємо у вхідний файл значення values[indexOfMin][indexOfMin == BUFF_1 ? pBuff1 : pBuff2]

ЯКЩО indexOfMin == BUFF_1, ТО pBuff1++, ІНАКШЕ pBuff2++

КІНЕЦЬ ЦИКЛУ

КІНЕЦЬ merge

- Програмна реалізація алгоритму
  - Вихідний код
    - Straight Merge

```java
package ua.algorithms.lab1.mvc.model;

import ua.algorithms.lab1.exception.*;
import ua.algorithms.lab1.property.Property;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.util.Objects;
import java.util.Properties;
import java.util.concurrent.TimeUnit;

import static ua.algorithms.lab1.utils.Utils.fromRef;

public class StraightMerge implements Model {

    private final RandomAccessFile outputAccess;
    private final RandomAccessFile buff1Access;
    private final RandomAccessFile buff2Access;
    private final long sourceLength;
    private byte[] buff;
    private int buffPointer;
    private static final File output;
    private static final File buff1;
    private static final File buff2;
    private static final int DEFAULT_BUFF_SIZE;
    private static final int MAX_ARRAY_SIZE =
Integer.MAX_VALUE - 8;
    private static final int BUFF_1 = 0;
```

```
    private static final int BUFF_2 = 1;

    private StraightMerge(
            RandomAccessFile outputAccess,
            RandomAccessFile buff1Access,
            RandomAccessFile buff2Access,
            long sourceLength
    ) {
        this.outputAccess = outputAccess;
        this.buff1Access = buff1Access;
        this.buff2Access = buff2Access;
        this.sourceLength = sourceLength;
        this.buff = new byte[0];
    }

    static {
        Properties properties =
Property.getInstance().getProperties();
        buff1 = new
File(properties.getProperty("default.path.buff1"));
        buff2 = new
File(properties.getProperty("default.path.buff2"));
        output = new
File(properties.getProperty("default.path.output"));
        DEFAULT_BUFF_SIZE =
Integer.parseInt(properties.getProperty("default.buffer.siz
e"));
    }

    public static StraightMerge getInstance(File source)
throws CopyFileException, FileNotFoundException {
        copySourceFile(source);
        RandomAccessFile outputAccess =
```

```
openConnection(output);
        createFile(buff1);
        RandomAccessFile buff1Access =
openConnection(buff1);
        createFile(buff2);
        RandomAccessFile buff2Access =
openConnection(buff2);
        long sourceLength = source.length();
        return new StraightMerge(outputAccess, buff1Access,
buff2Access, sourceLength);
    }

    private static void createFile(File file) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            throw new FileCreateException(e);
        }
    }

    private static void copySourceFile(File source) throws
CopyFileException {
        Path original = source.toPath();
        Path copied = StraightMerge.output.toPath();
        try {
            Files.copy(original, copied,
StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {
            throw new CopyFileException(e);
        }
    }

    private static RandomAccessFile openConnection(File
```

```
file) throws FileNotFoundException {
        try {
            return new RandomAccessFile(file, "rw");
        } catch (FileNotFoundException e) {
            throw new
FileNotFoundException(e.getMessage());
        }
    }

    @Override
    public void sort() throws FileNotFoundException,
CloseConnectionException, FileAccessException {
        long digitsNum = 1;
        if (sourceLength / Integer.BYTES <= 1) {
            System.out.println("The data is already
sorted");
            return;
        }
        try {
            long start = System.nanoTime();
            while (digitsNum < sourceLength /
Integer.BYTES) {
                distribution(digitsNum);
                merge(digitsNum);
                digitsNum *= 2;
            }
            long finish = System.nanoTime();
            System.out.printf("Sorting took %d seconds%n",
TimeUnit.NANOSECONDS.toSeconds(finish - start));
            close();
        } catch (FileAccessException e) {
            closeWithTryCatch();
            throw new FileAccessException(e);
```

```
        } catch (IOException e) {
            closeWithTryCatch();
        }
    }

    private void closeWithTryCatch() throws
CloseConnectionException {
        try {
            close();
        } catch (IOException e) {
            throw new CloseConnectionException("Can't close
connection", e);
        }
    }

    private void flush() throws WriteToFileException {
        try {
            if (buff.length > 0) {
                outputAccess.write(buff);
                buffPointer = 0;
            }
        } catch (IOException e) {
            throw new
WriteToFileException(String.format("%s (Failed while
flushing)", output.getPath()), e);
        }
    }

    private void merge(long digitsNum) throws
FileAccessException {
        long[] globalPointers = new long[2];
        int[] localPointers = new int[2];
        Integer[][] values = initializeArr(digitsNum);
```

```
        for (long i = 0; i < sourceLength / Integer.BYTES;
i++) {
            int index;
            long endOfGroupIndex;
            Integer minValue;
            if (buffPointer >= buff.length) {
                flush();
                if ((sourceLength - i * Integer.BYTES) >
DEFAULT_BUFF_SIZE) {
                    buff = new byte[DEFAULT_BUFF_SIZE];
                } else {
                    buff = new byte[(int) (sourceLength - i
* Integer.BYTES)];
                }
            }
            if (Objects.isNull(values[BUFF_1])) {
                index = BUFF_2;
                minValue =
values[BUFF_2][localPointers[BUFF_2]];
            } else if (Objects.isNull(values[BUFF_2])) {
                index = BUFF_1;
                minValue =
values[BUFF_1][localPointers[BUFF_1]];
            } else {
                index = findIndexOfMin(values,
localPointers);
                minValue =
values[index][localPointers[index]];
            }
            addToBuff(minValue);
            endOfGroupIndex = ((globalPointers[index] /
digitsNum) + 1) * digitsNum;
            localPointers[index]++;
```

```
            globalPointers[index]++;
            if (localPointers[index] >=
values[index].length) {
                if (globalPointers[index] < endOfGroupIndex
                        && globalPointers[index] <
getFileLength(index == BUFF_1 ? buff1Access : buff2Access)
/ Integer.BYTES) {
                    values[index] = readPart(index ==
BUFF_1 ? buff1Access : buff2Access, digitsNum,
globalPointers[index]);
                    localPointers[index] = 0;
                } else {
                    values[index] = null;
                }
            }
            if (Objects.isNull(values[BUFF_1]) &&
Objects.isNull(values[BUFF_2])) {
                values[BUFF_1] = readPart(buff1Access,
digitsNum, globalPointers[BUFF_1]);
                values[BUFF_2] = readPart(buff2Access,
digitsNum, globalPointers[BUFF_2]);
                localPointers[BUFF_1] = 0;
                localPointers[BUFF_2] = 0;
            }
        }
        flush();
        try {
            outputAccess.seek(0);
        } catch (IOException e) {
            throw new FileAccessException(String.format("%s
(Failed to seek)", output.getPath()), e);
        }
        clearBuffers();
```

```
    }

    private void clearBuffers() throws FileAccessException
{
        try {
            buff1Access.setLength(0);
        } catch (IOException e) {
            throw new FileAccessException(String.format("%s
(Failed to set length)", buff1.getPath()), e);
        }
        try {
            buff2Access.setLength(0);
        } catch (IOException e) {
            throw new FileAccessException(String.format("%s
(Failed to set length)", buff2.getPath()), e);
        }
    }

    private static int findIndexOfMin(Integer[][] values,
int[] pointers) {
        int index = 0;
        while (values[index] == null) index++;
        Integer minValue = values[index][pointers[index]];
        for (int i = index + 1; i < values.length; i++) {
            if (values[i] != null &&
minValue.compareTo(values[i][pointers[i]]) > 0) {
                minValue = values[i][pointers[i]];
                index = i;
            }
        }
        return index;
    }
```

```
    private void addToBuff(int value) {
        buff[buffPointer++] = (byte) (value >> 24);
        buff[buffPointer++] = (byte) (value >> 16);
        buff[buffPointer++] = (byte) (value >> 8);
        buff[buffPointer++] = (byte) value;
    }

    private Integer[][] initializeArr(long digitsNum)
throws FileAccessException {
        Integer[][] values = new Integer[2][];
        try {
            values[0] = readPart(buff1Access, digitsNum,
0);
        } catch (FetchFileLengthException |
ReadFromFileException e) {
            throw new FileAccessException(String.format("%s
%s", buff1.getPath(), e.getMessage()), e);
        }
        try {
            values[1] = readPart(buff2Access, digitsNum,
0);
        } catch (FetchFileLengthException |
ReadFromFileException e) {
            throw new FileAccessException(String.format("%s
%s", buff2.getPath(), e.getMessage()), e);
        }
        return values;
    }

    private Integer[] readPart(RandomAccessFile raf, long
digitsNum, long ptr)
            throws FetchFileLengthException,
ReadFromFileException {
```

```
        if (ptr < getFileLength(raf) / Integer.BYTES) {
            byte[] buff;
            long currGroupEndIndex = (((ptr / digitsNum) +
1) * digitsNum);
            long toRead = Math.min((getFileLength(raf) /
Integer.BYTES - ptr), currGroupEndIndex - ptr) *
Integer.BYTES;
            if (toRead >= DEFAULT_BUFF_SIZE) {
                buff = new byte[DEFAULT_BUFF_SIZE];
            } else {
                buff = new byte[(int) toRead];
            }
            try {
                raf.read(buff);
            } catch (IOException e) {
                throw new ReadFromFileException("(Failed to
read from file)", e);
            }
            return fromRef(buff);
        } else {
            return null;
        }
    }

    private static long getFileLength(RandomAccessFile raf)
throws FetchFileLengthException {
        try {
            return raf.length();
        } catch (IOException e) {
            throw new FetchFileLengthException("(Failed to
get file's length)", e);
        }
    }
```

```
    private void distribution(long digitsNum) throws
FileAccessException {
        for (long i = 0; i < Math.ceil((sourceLength /
(double) Integer.BYTES) / (double) digitsNum); i++) {
            long remainder = sourceLength - (i * digitsNum)
* Integer.BYTES;
            long bytesToRead = digitsNum * Integer.BYTES;
            long size = Math.min(remainder, bytesToRead);
            if (i % 2 == 0) {
                try {
                    moveGroup(buff1Access, size);
                } catch (WriteToFileException e) {
                    throw new
WriteToFileException(String.format("%s %s",
buff1.getPath(), e.getMessage()), e);
                }
            } else {
                try {
                    moveGroup(buff2Access, size);
                } catch (WriteToFileException e) {
                    throw new
WriteToFileException(String.format("%s %s",
buff2.getPath(), e.getMessage()), e);
                }
            }
        }
        moveFilePointersToStart();
    }

    private void moveFilePointersToStart() throws
FileAccessException {
        try {
```

```
            buff1Access.seek(0);
            buff2Access.seek(0);
            outputAccess.seek(0);
        } catch (IOException e) {
            throw new FileAccessException("(Failed to seek
file position)", e);
        }
    }

    private void moveGroup(RandomAccessFile buff, long
size) throws ReadFromFileException, WriteToFileException {
        try {
            if (size > MAX_ARRAY_SIZE) {
                long bytesWritten = 0;
                for (int i = 0; i < Math.ceil(size /
(double) MAX_ARRAY_SIZE); i++) {
                    long remainder = size - bytesWritten;
                    int length = remainder > MAX_ARRAY_SIZE
? MAX_ARRAY_SIZE : (int) remainder;
                    writeGroup(buff,
readGroup(outputAccess, length));
                    bytesWritten += length;
                }
            } else {
                writeGroup(buff, readGroup(outputAccess,
(int) size));
            }
        } catch (ReadFromFileException e) {
            throw new
ReadFromFileException(String.format("%s %s",
output.getPath(), e.getMessage()), e);
        }
    }
```

```
    private void writeGroup(RandomAccessFile dest, byte[]
group) throws WriteToFileException {
        try {
            dest.write(group);
        } catch (IOException e) {
            throw new WriteToFileException("(Failed while
writing group to file)", e);
        }
    }

    private byte[] readGroup(RandomAccessFile src, int
length) throws ReadFromFileException {
        byte[] group = new byte[length];
        try {
            src.read(group);
        } catch (IOException e) {
            throw new ReadFromFileException("(Failed while
reading group from file)", e);
        }
        return group;
    }

    public void close() throws IOException {
        outputAccess.close();
        buff1Access.close();
        buff2Access.close();
        buff1.delete();
        buff2.delete();
    }
}
```

- Improved Straight Merge

```
package ua.algorithms.lab1.mvc.model;

import ua.algorithms.lab1.exception.*;
import ua.algorithms.lab1.property.Property;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.TimeUnit;

import static ua.algorithms.lab1.utils.Utils.*;

public class ImprovedStraightMerge implements Model {
    private final RandomAccessFile outputAccess;
    private final RandomAccessFile sourceAccess;
    private final File source;
    private final File output;
    private final long sourceLength;
    private byte[] buff;
    private int buffPointer;
    private final int chunksPartSize;
    private final Map<Integer, File> indexChunkMap;
    private final Map<Integer, RandomAccessFile>
indexChunkAccessMap;
    private static final String CHUNKS_PATH;
    private static final String OUTPUT_PATH;
```

```
    private static final int CHUNK_BYTE_SIZE;

    static {
        Properties properties =
Property.getInstance().getProperties();
        CHUNKS_PATH =
properties.getProperty("default.path.chunks");
        OUTPUT_PATH =
properties.getProperty("default.path.output");
        CHUNK_BYTE_SIZE =
Integer.parseInt(properties.getProperty("default.chunk.size
"));
    }

    private ImprovedStraightMerge(
            RandomAccessFile sourceAccess,
            RandomAccessFile outputAccess,
            int chunksPartSize,
            long sourceLength,
            File source,
            File output
    ) {
        this.source = source;
        this.output = output;
        this.sourceLength = sourceLength;
        this.sourceAccess = sourceAccess;
        this.outputAccess = outputAccess;
        this.chunksPartSize = chunksPartSize;
        this.buff = new byte[chunksPartSize];
        this.indexChunkMap = new HashMap<>();
        this.indexChunkAccessMap = new HashMap<>();
    }
```

```
    public static ImprovedStraightMerge getInstance(File
source)
            throws FileNotFoundException,
FileAccessException {
        long sourceLength;
        RandomAccessFile sourceAccess, outputAccess;
        File output = new File(OUTPUT_PATH);
        try {
            sourceAccess = new RandomAccessFile(source,
"r");
            sourceLength = getFileLength(sourceAccess);
        } catch (FileNotFoundException e) {
            throw new
FileNotFoundException(String.format("%s", e.getMessage()));
        } catch (FetchFileLengthException e) {
            throw new
FetchFileLengthException(String.format("%s %s",
source.getPath(), e.getMessage()), e);
        }
        try {
            outputAccess = new RandomAccessFile(output,
"rw");
            try {
                outputAccess.setLength(0);
            } catch (IOException e) {
                throw new
FileAccessException(String.format("%s (Failed to set
length)", OUTPUT_PATH), e);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Output file doesn't exist,
it'll be created");
            try {
```

```
                output.createNewFile();
            } catch (IOException ex) {
                throw new
FileCreateException(String.format("%s (Can't create file)",
output.getPath()), ex);
            }
            outputAccess = new RandomAccessFile(output,
"rw");
        }
        int chunkPartSize = ((int) (CHUNK_BYTE_SIZE /
(Math.ceil(source.length() / (double) CHUNK_BYTE_SIZE) +
1)) / Integer.BYTES) * Integer.BYTES;
        return new ImprovedStraightMerge(sourceAccess,
outputAccess, chunkPartSize, sourceLength, source, output);
    }

    @Override
    public void sort()
            throws FileNotFoundException,
            CloseConnectionException,
            FileAccessException {
        if (sourceLength / Integer.BYTES <= 1) {
            System.out.println("The data is already
sorted");
            return;
        }
        try {
            long start = System.nanoTime();
            if (sourceLength <= CHUNK_BYTE_SIZE) {
                sortSmallData();
            } else {
                sortChunks();
                merge();
```

```
                flush();
            }
            long finish = System.nanoTime();
            System.out.printf("Sorting took %d seconds\n",
TimeUnit.NANOSECONDS.toSeconds(finish - start));
            close();
        } catch (FileAccessException e) {
            closeWithTryCatch();
            throw new FileAccessException(e);
        } catch (FileNotFoundException e) {
            closeWithTryCatch();
            throw new
FileNotFoundException(e.getMessage());
        } catch (IOException e) {
            closeWithTryCatch();
        }
    }

    private void sortSmallData() throws
ReadFromFileException, WriteToFileException {
        byte[] bytes = new byte[(int) sourceLength];
        try {
            sourceAccess.read(bytes);
        } catch (IOException e) {
            throw new
ReadFromFileException(String.format("%s (Failed while
reading)", source.getPath()), e);
        }
        int[] ints = from(bytes);
        quickSort(ints, 0, ints.length - 1);
        byte[] sorted = from(ints);
        try {
            outputAccess.write(sorted);
```

```
        } catch (IOException e) {
            throw new
WriteToFileException(String.format("%s (Failed while
writing)", output.getPath()), e);
        }
    }

    private void closeWithTryCatch() throws
CloseConnectionException {
        try {
            close();
        } catch (IOException e) {
            throw new CloseConnectionException("Can't close
connection", e);
        }
    }

    private void merge() throws FileAccessException {
        long length = sourceLength / Integer.BYTES;
        int[] localPointers = new
int[indexChunkMap.size()];
        long[] globalPointers = new
long[indexChunkMap.size()];
        Integer[][] values = initializeArr();
        for (long i = 0; i < length; i++) {
            int minValuesIndex = findIndexOfMin(values,
localPointers);
            Integer minValue =
values[minValuesIndex][localPointers[minValuesIndex]];
            if (buffPointer >= buff.length) {
                flush();
                if ((length * Integer.BYTES - i *
Integer.BYTES) > chunksPartSize) {
```

```
                    buff = new byte[chunksPartSize];
                } else {
                    buff = new byte[(int) (length *
Integer.BYTES - i * Integer.BYTES)];
                }
            }
            addToBuff(minValue);
            localPointers[minValuesIndex]++;
            globalPointers[minValuesIndex]++;
            if (localPointers[minValuesIndex] >=
values[minValuesIndex].length) {
                RandomAccessFile raf =
indexChunkAccessMap.get(minValuesIndex);
                if (globalPointers[minValuesIndex] <
(getFileLength(raf) / Integer.BYTES)) {
                    values[minValuesIndex] =
readPartOfChunk(raf, globalPointers[minValuesIndex]);
                    localPointers[minValuesIndex] = 0;
                } else {
                    values[minValuesIndex] = null;
                }
            }
        }
    }

    private static long getFileLength(RandomAccessFile raf)
throws FetchFileLengthException {
        try {
            return raf.length();
        } catch (IOException e) {
            throw new FetchFileLengthException("(Failed to
get file's length)", e);
        }
```

```
    }

    private void flush() throws WriteToFileException {
        try {
            writeToFile(outputAccess, buff);
        } catch (WriteToFileException e) {
            throw new
WriteToFileException(String.format("%s %s",
output.getPath(), e.getMessage()), e);
        }
        buffPointer = 0;
    }

    private static void writeToFile(RandomAccessFile raf,
byte[] toWrite) throws WriteToFileException {
        try {
            raf.write(toWrite);
        } catch (IOException e) {
            throw new WriteToFileException("(Can't write to
file)", e);
        }
    }

    private void addToBuff(int value) {
        buff[buffPointer++] = (byte) (value >> 24);
        buff[buffPointer++] = (byte) (value >> 16);
        buff[buffPointer++] = (byte) (value >> 8);
        buff[buffPointer++] = (byte) value;
    }


    private static int findIndexOfMin(Integer[][] values,
int[] pointers) {
```

```
            int index = 0;
            while (values[index] == null) index++;
            Integer minValue = values[index][pointers[index]];
            for (int i = index + 1; i < values.length; i++) {
                if (values[i] != null &&
minValue.compareTo(values[i][pointers[i]]) > 0) {
                    minValue = values[i][pointers[i]];
                    index = i;
                }
            }
            return index;
        }

        private Integer[][] initializeArr() throws
FileAccessException {
            int length = indexChunkMap.size();
            Integer[][] values = new Integer[length][];
            for (int i = 0; i < length; i++) {
                RandomAccessFile raf =
indexChunkAccessMap.get(i);
                try {
                    values[i] = readPartOfChunk(raf, 0);
                } catch (FileAccessException e) {
                    File badFile = indexChunkMap.get(i);
                    throw new
FileAccessException(String.format("%s %s",
badFile.getPath(), e.getMessage()), e);
                }
            }
            return values;
        }

        private Integer[] readPartOfChunk(RandomAccessFile raf,
```

```
long prt)
            throws FetchFileLengthException,
ReadFromFileException {
        byte[] buff;
        if (((getFileLength(raf) / Integer.BYTES) - prt) >
chunksPartSize / Integer.BYTES) {
            buff = new byte[chunksPartSize];
        } else {
            buff = new byte[(int) (((getFileLength(raf) /
Integer.BYTES) - prt) * Integer.BYTES)];
        }
        try {
            raf.read(buff);
        } catch (IOException e) {
            throw new ReadFromFileException("(Failed to
read bytes from file)", e);
        }
        return fromRef(buff);
    }

    public void sortChunks() throws FileAccessException,
FileNotFoundException {
        int chunksNumber = (int) Math.ceil(sourceLength /
(double) CHUNK_BYTE_SIZE);
        for (int i = 0; i < chunksNumber; i++) {
            int bytesToRead;
            if (sourceLength - (long) i * CHUNK_BYTE_SIZE >
CHUNK_BYTE_SIZE)
                bytesToRead = CHUNK_BYTE_SIZE;
            else
                bytesToRead = (int) (sourceLength - i *
CHUNK_BYTE_SIZE);
            int[] chunk = readChunk(bytesToRead);
```

```
                quickSort(chunk, 0, chunk.length - 1);
                writeChunk(chunk);
            }
        }

        private void writeChunk(int[] chunk)
                throws FileNotFoundException,
    FileAccessException {
            int index = indexChunkMap.size();
            String chunkName = String.format("Chunk%d", index);
            File chunkBinFile = new File(CHUNKS_PATH,
    chunkName);
            createFile(chunkBinFile);
            indexChunkMap.put(index, chunkBinFile);
            RandomAccessFile chunkFileAccess =
    openChunkConnection(chunkBinFile);
            indexChunkAccessMap.put(index, chunkFileAccess);
            byte[] bytesChuck = from(chunk);
            try {
                writeToFile(chunkFileAccess, bytesChuck);
            } catch (WriteToFileException e) {
                throw new
    WriteToFileException(String.format("%s %s",
    chunkBinFile.getPath(), e.getMessage()), e);
            }
            try {
                chunkFileAccess.seek(0);
            } catch (IOException e) {
                throw new FileAccessException(String.format("%s
    (Can't seek position 0)", chunkBinFile.getPath()), e);
            }
        }
```

```
    private void createFile(File file) throws
FileCreateException {
        try {
            file.createNewFile();
        } catch (IOException e) {
            throw new FileCreateException(String.format("%s
(%s)", file.getPath(), e.getMessage()), e);
        }
    }

    private RandomAccessFile openChunkConnection(File file)
throws FileNotFoundException {
        try {
            return new RandomAccessFile(file, "rw");
        } catch (FileNotFoundException e) {
            throw new
FileNotFoundException(String.format("%s (%s)",
file.getPath(), e.getMessage()));
        }
    }

    private int[] readChunk(int chunkSize) throws
ReadFromFileException {
        return from(readChunkBytes(chunkSize));
    }

    private byte[] readChunkBytes(int chunkSize) throws
ReadFromFileException {
        byte[] chunk = new byte[chunkSize];
        try {
            sourceAccess.read(chunk);
        } catch (IOException e) {
            throw new
```

```
ReadFromFileException(String.format("%s (Exception while
reading chunk from file)", source.getPath()), e);
            }
            return chunk;
        }

        public void close() throws IOException {
            sourceAccess.close();
            outputAccess.close();
            for (RandomAccessFile raf :
indexChunkAccessMap.values()) raf.close();
            for (File f : indexChunkMap.values()) f.delete();
        }
}
```

♦ Висновок

На лабораторній роботі було декомпозовано задачу на такі етапи: написання псевдокоду алгоритму, програмна реалізація алгоритму зовнішнього сортування “Пряме злиття”, програмна реалізація модифікації алгоритму зовнішнього сортування “Пряме злиття”, порівняння двох алгоритмів у висновку. При порівнянні двох алгоритмів було досліджено, що немодифікована версія сортує невпорядковану множину цілих чисел значно довже ніж модифікована. 10 мегабайт даних немодифікований алгоритм сортує в середньому за 1 хвилину, в той час як у модифікованого це займає не більше 1 секунди. Такий значний приріст продуктивності було досягнуто завдяки попередньому впорядкуванню серій елементів по 100 мегабайт, а також завдяки зменшенню кількості звертань до системного ядра, що було досягнуто завдяки створенню вихідного буфера, у який записувалися результуючі частини і у разі його заповнення викликався метод flush(), який записував усі дані в результуючий файл, виконуючи лише одне звернення до ядра.