

**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 No 1 з дисципліни

«Проектування алгоритмів»

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

**Виконав(ла)**

ІП-22, Андрєєва Уляна Андріївна

(шифр, прізвище, ім'я, по батькові)

**Перевірів**

Ахаладзе Ілля Елдарійович

(шифр, прізвище, ім'я, по батькові)

Київ 2023

## ЗМІСТ

МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....	5
ЗАВДАННЯ .....	6
ВИКОНАННЯ .....	7
1. 3.1 ПСЕВДОКОД АЛГОРИТМУ .....	7
2. 3.2 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ.....	10
3.2.1 Вихідний код .....	10
ВИСНОВОК .....	23
КРИТЕРІЇ ОЦІНЮВАННЯ .....	15

## **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. 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.

***№ 1 Алгоритм сортування - пряме злиття***

## 3 ВИКОНАННЯ

### 3.1 ПСЕВДОКОД АЛГОРИТМУ

```
function externalMergeSort(inputFile, outputFile, chunkSize):
```

```
    splitInputFile(inputFile, chunkSize)
```

```
    mergeSortedChunks(outputFile)
```

```
function splitInputFile(inputFile, chunkSize):
```

```
    chunkNum = 0
```

```
    chunk = create empty array of integers of size chunkSize
```

```
    index = 0
```

```
    open inputFile as input
```

```
    while not end of inputFile:
```

```
        line = read next line from input
```

```
        chunk[index] = parse line as integer
```

```
        index = index + 1
```

```
    if index == chunkSize:
```

```
        sort chunk
```

```
        writeChunk(chunk, "chunk_" + chunkNum + ".txt")
```

```
        chunkNum = chunkNum + 1
```

```
        index = 0
```

```
    if index > 0:
```

```
        sort first index elements of chunk
```

```
        writeChunk(chunk, "chunk_" + chunkNum + ".txt")
```

```

function writeChunk(chunk, fileName):
    open fileName as output

    for num in chunk:
        write num to output

function mergeSortedChunks(outputFile):
    chunkFiles = list of files in current directory that start with "chunk_"
    minValues = array of integers of size length(chunkFiles)
    readers = array of BufferedReader of size length(chunkFiles)

    open outputFile as output

    for i = 0 to length(chunkFiles) - 1:
        readers[i] = open chunkFiles[i] for reading
        minValues[i] = parse first line of readers[i]

    while true:
        minIndex = findMinIndex(minValues)
        if minIndex == -1:
            break

        write minValues[minIndex] to output
        nextLine = read next line from readers[minIndex]

```

```
if nextLine == null:

    minValues[minIndex] = MAX_VALUE

else:

    minValues[minIndex] = parse nextLine
```

```
function findMinIndex(array):
```

```
    min = MAX_VALUE
```

```
    minIndex = -1
```

```
    for i = 0 to length(array) - 1:
```

```
        if array[i] < min:
```

```
            min = array[i]
```

```
            minIndex = i
```

```
    return minIndex
```

## 3.2 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ

### 3.2.1 Вихідний код

#### **ExternalMergeSortFile**

```
import java.io.*;
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class ExternalMergeSortFile {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the name of the file: ");
        String fileName = scanner.nextLine();

        long fileSizeInBytes = getFileSizeFromUser(scanner);

        try {
            long startTime = System.currentTimeMillis();
            createAndPopulateLargeFile(fileName, fileSizeInBytes);
            long endTime = System.currentTimeMillis();
            System.out.println("File was populated successfully - " + fileName);
            long elapsedTime = endTime - startTime;
            System.out.println("File creation time: " + (elapsedTime / 60000) + " minutes");

            startTime = System.currentTimeMillis();
            externalMergeSort(fileName, "sorted_" + fileName, 1000000);
            endTime = System.currentTimeMillis();
            System.out.println("Sorting was ended.");
            elapsedTime = endTime - startTime;
            System.out.println("Sorting time: " + (endTime - startTime) + " ms");
            System.out.println("Sorting time: " + (elapsedTime / 60000) + " minutes");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```

    }
}

private static long getFileSizeFromUser(Scanner scanner) {
    long fileSizeInBytes;
    while (true) {
        System.out.print("Enter the size of the file in mb (integer): ");
        try {
            String input = scanner.next();
            fileSizeInBytes = Long.parseLong(input) * 1024L * 1024L;

            if (fileSizeInBytes <= 0) {
                throw new IllegalArgumentException("The size of the file should be bigger than
zero");
            }

            break;
        } catch (NumberFormatException e) {
            System.out.println("Invalid input. Please enter a valid integer.");
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
    return fileSizeInBytes;
}

```

```

public static void createAndPopulateLargeFile(String fileName, long fileSizeInBytes)
throws IOException {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {
        long totalNumbers = fileSizeInBytes / 4;
        Random random = new Random();
        for (long i = 0; i < totalNumbers; i++) {
            int randomNumber = random.nextInt();

```

```

        bw.write(Integer.toString(randomNumber));
        bw.newLine();
    }
}

public static void externalMergeSort(String inputFile, String outputFile, int chunkSize)
throws IOException {
    ChunkFileSplitter.splitInputFile(inputFile, chunkSize);
    ChunkFileMerger.mergeSortedChunks(outputFile);
}

class ChunkFileSplitter {

    public static void splitInputFile(String inputFile, int chunkSize) throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
            int[] chunk = new int[chunkSize];
            int index = 0;
            int chunkNum = 0;

            long startTime = System.currentTimeMillis();

            String line;
            while ((line = br.readLine()) != null) {
                chunk[index++] = Integer.parseInt(line);

                if (index == chunkSize) {
                    sortAndWriteChunk(chunk, "chunk_" + chunkNum + ".txt");
                    chunkNum++;
                    index = 0;
                }
            }
        }
    }
}

```

```

        if (index > 0) {
            sortAndWriteChunk(chunk, "chunk_" + chunkNum + ".txt");
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Splitting time: " + (endTime - startTime) + " ms");
    }
}

private static void sortAndWriteChunk(int[] chunk, String fileName) throws IOException
{
    Arrays.sort(chunk);
    writeChunkToFile(chunk, fileName);
}

private static void writeChunkToFile(int[] chunk, String fileName) throws IOException {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {
        for (int num : chunk) {
            bw.write(Integer.toString(num));
            bw.newLine();
        }
    }
}

class ChunkFileMerger {
    public static void mergeSortedChunks(String outputFile) throws IOException {
        File[] chunkFiles = findChunkFiles();
        int[] minValues = new int[chunkFiles.length];
        BufferedReader[] readers = openReaders(chunkFiles);
        BufferedWriter bw = openWriter(outputFile);

        readInitialValues(minValues, readers);

        long startTime = System.currentTimeMillis();

```

```

mergeChunks(minValues, readers, bw);

long endTime = System.currentTimeMillis();
System.out.println("Merging time: " + (endTime - startTime) + " ms");

closeResources(readers, bw);
deleteChunkFiles(chunkFiles);
}

private static BufferedWriter openWriter(String outputFile) throws IOException {
    return new BufferedWriter(new FileWriter(outputFile));
}

private static void mergeChunks(int[] minValues, BufferedReader[] readers,
BufferedWriter bw) throws IOException {
    while (true) {
        int minIndex = findMinIndex(minValues);
        if (minIndex == -1) {
            break;
        }
        writeMinValue(minValues[minIndex], bw);
        updateMinValue(minIndex, minValues, readers);
    }
}

private static File[] findChunkFiles() {
    return new File(".").listFiles((dir, name) -> name.startsWith("chunk_"));
}

private static BufferedReader[] openReaders(File[] chunkFiles) throws
FileNotFoundException {
    BufferedReader[] readers = new BufferedReader[chunkFiles.length];
    for (int i = 0; i < chunkFiles.length; i++) {
        readers[i] = new BufferedReader(new FileReader(chunkFiles[i]));
    }
}

```

```

    }
    return readers;
}

```

```

private static void readInitialValues(int[] minValues, BufferedReader[] readers) throws
IOException {
    for (int i = 0; i < readers.length; i++) {
        minValues[i] = Integer.parseInt(readers[i].readLine());
    }
}

```

```

private static int findMinIndex(int[] array) {
    int min = Integer.MAX_VALUE;
    int minIndex = -1;
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
            minIndex = i;
        }
    }
    return minIndex;
}

```

```

private static void writeMinValue(int minValue, BufferedWriter bw) throws IOException
{
    bw.write(Integer.toString(minValue));
    bw.newLine();
}

```

```

private static void updateMinValue(int minIndex, int[] minValues, BufferedReader[]
readers) throws IOException {
    String nextLine = readers[minIndex].readLine();
    if (nextLine == null) {
        minValues[minIndex] = Integer.MAX_VALUE;
    }
}

```

```

    } else {
        minValues[minIndex] = Integer.parseInt(nextLine);
    }
}

private static void closeResources(BufferedReader[] readers, BufferedWriter bw) throws
IOException {
    bw.close();
    for (BufferedReader reader : readers) {
        reader.close();
    }
}

private static void deleteChunkFiles(File[] chunkFiles) {
    for (File file : chunkFiles) {
        try {
            if (file.delete()) {
                System.out.println("File deleted successfully: " + file.getName());
            } else {
                System.err.println("Failed to delete file: " + file.getName());
            }
        } catch (SecurityException e) {
            System.err.println("SecurityException while deleting file: " + file.getName());
            e.printStackTrace();
        }
    }
}
}

```

## DefaultExternalMergeSortFile

```
import java.io.*;
import java.util.*;

interface FileSplitter {
    void splitInputFile(String inputFile, int chunkSize) throws IOException;
}

interface FileMerger {
    void mergeSortedChunks(String outputFile) throws IOException;
}

class DefaultFileSplitter implements FileSplitter {
    @Override
    public void splitInputFile(String inputFile, int chunkSize) throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
            int[] chunk = new int[chunkSize];
            int index = 0;
            int chunkNum = 0;

            long startTime = System.currentTimeMillis();

            String line;
            while ((line = br.readLine()) != null) {
                chunk[index++] = Integer.parseInt(line);

                if (index == chunkSize) {
                    sortAndWriteChunk(chunk, "chunk_" + chunkNum + ".txt");
                    chunkNum++;
                    index = 0;
                }
            }
        }
    }
}
```

```

        if (index > 0) {
            sortAndWriteChunk(chunk, "chunk_" + chunkNum + ".txt");
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Splitting time: " + (endTime - startTime) + " ms");
    }
}

private void sortAndWriteChunk(int[] chunk, String fileName) throws IOException {
    Arrays.sort(chunk);
    writeChunkToFile(chunk, fileName);
}

private void writeChunkToFile(int[] chunk, String fileName) throws IOException {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {
        for (int num : chunk) {
            bw.write(Integer.toString(num));
            bw.newLine();
        }
    }
}

class DefaultFileMerger implements FileMerger {
    @Override
    public void mergeSortedChunks(String outputFile) throws IOException {
        File[] chunkFiles = findChunkFiles();
        int[] minValues = new int[chunkFiles.length];
        BufferedReader[] readers = openReaders(chunkFiles);
        BufferedWriter bw = openWriter(outputFile);

        readInitialValues(minValues, readers);
    }
}

```



```

    long startTime = System.currentTimeMillis();

    mergeChunks(minValues, readers, bw);

    long endTime = System.currentTimeMillis();
    System.out.println("Merging time: " + (endTime - startTime) + " ms");

    closeResources(readers, bw);
    deleteChunkFiles(chunkFiles);
}

private void closeResources(BufferedReader[] readers, BufferedWriter bw) throws
IOException {
    bw.close();
    for (BufferedReader reader : readers) {
        reader.close();
    }
}

private BufferedWriter openWriter(String outputFile) throws IOException {
    return new BufferedWriter(new FileWriter(outputFile));
}

private void mergeChunks(int[] minValues, BufferedReader[] readers, BufferedWriter bw)
throws IOException {
    while (true) {
        int minIndex = findMinIndex(minValues);
        if (minIndex == -1) {
            break;
        }
        writeMinValue(minValues[minIndex], bw);
        updateMinValue(minIndex, minValues, readers);
    }
}

```

```

private File[] findChunkFiles() {
    return new File(".").listFiles((dir, name) -> name.startsWith("chunk_"));
}

private void deleteChunkFiles(File[] chunkFiles) {
    for (File file : chunkFiles) {
        try {
            if (file.delete()) {
                System.out.println("File deleted successfully: " + file.getName());
            } else {
                System.err.println("Failed to delete file: " + file.getName());
            }
        } catch (SecurityException e) {
            System.err.println("SecurityException while deleting file: " + file.getName());
            e.printStackTrace();
        }
    }
}

private BufferedReader[] openReaders(File[] chunkFiles) throws FileNotFoundException
{
    BufferedReader[] readers = new BufferedReader[chunkFiles.length];
    for (int i = 0; i < chunkFiles.length; i++) {
        readers[i] = new BufferedReader(new FileReader(chunkFiles[i]));
    }
    return readers;
}

private void readInitialValues(int[] minValues, BufferedReader[] readers) throws
IOException {
    for (int i = 0; i < readers.length; i++) {
        minValues[i] = Integer.parseInt(readers[i].readLine());
    }
}

```

```

private int findMinIndex(int[] array) {
    int min = Integer.MAX_VALUE;
    int minIndex = -1;
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
            minIndex = i;
        }
    }
    return minIndex;
}

```

```

private void writeMinValue(int minValue, BufferedWriter bw) throws IOException {
    bw.write(Integer.toString(minValue));
    bw.newLine();
}

```

```

private void updateMinValue(int minIndex, int[] minValues, BufferedReader[] readers)
throws IOException {
    String nextLine = readers[minIndex].readLine();
    if (nextLine == null) {
        minValues[minIndex] = Integer.MAX_VALUE;
    } else {
        minValues[minIndex] = Integer.parseInt(nextLine);
    }
}
}

```

```

public class DefaultExternalMergeSortFile {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the name of the file: ");
        String fileName = scanner.nextLine();
    }
}

```

```

long fileSizeInBytes = getFileSizeFromUser(scanner);

try {
    long startTime = System.currentTimeMillis();
    createAndPopulateLargeFile(fileName, fileSizeInBytes);
    long endTime = System.currentTimeMillis();
    System.out.println("File was populated successfully - " + fileName);
    long elapsedTime = endTime - startTime;
    System.out.println("File creation time: " + (elapsedTime / 60000) + " minutes");

    startTime = System.currentTimeMillis();
    ExternalFileSorter externalFileSorter = new ExternalFileSorter();
    externalFileSorter.externalMergeSort(fileName, "sorted_" + fileName, 1000000);
    endTime = System.currentTimeMillis();
    System.out.println("Sorting was ended.");
    elapsedTime = endTime - startTime;
    System.out.println("Sorting time: " + (endTime - startTime) + " ms");
    System.out.println("Sorting time: " + (elapsedTime / 60000) + " minutes");
} catch (IOException e) {
    e.printStackTrace();
}

}

private static long getFileSizeFromUser(Scanner scanner) {
    long fileSizeInBytes;
    while (true) {
        System.out.print("Enter the size of the file in mb (integer): ");
        try {
            String input = scanner.next();
            fileSizeInBytes = Long.parseLong(input) * 1024L * 1024L;

            if (fileSizeInBytes <= 0) {

```

```

        throw new IllegalArgumentException("The size of the file should be bigger than
zero");
    }

```

```

        break;
    } catch (NumberFormatException e) {
        System.out.println("Invalid input. Please enter a valid integer.");
    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }
}
return fileSizeInBytes;
}

```

```

public static void createAndPopulateLargeFile(String fileName, long fileSizeInBytes)
throws IOException {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {
        long totalNumbers = fileSizeInBytes / 4;
        Random random = new Random();
        for (long i = 0; i < totalNumbers; i++) {
            int randomNumber = random.nextInt();
            bw.write(Integer.toString(randomNumber));
            bw.newLine();
        }
    }
}
}

```

```

class ExternalFileSorter {
    private final FileSplitter fileSplitter;
    private final FileMerger fileMerger;

    public ExternalFileSorter() {
        this.fileSplitter = new DefaultFileSplitter();
    }
}

```

```

        this.fileMerger = new DefaultFileMerger();
    }

    public void externalMergeSort(String inputFile, String outputFile, int chunkSize) throws
IOException {
        fileSplitter.splitInputFile(inputFile, chunkSize);
        fileMerger.mergeSortedChunks(outputFile);
    }
}

```

## **Main**

```

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Choose an option:");
        System.out.println("1. Use DefaultExternalMergeSortFile");
        System.out.println("2. Use ExternalMergeSortFile");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                DefaultExternalMergeSortFile.main(args);
                break;
            case 2:
                ExternalMergeSortFile.main(args);
                break;
            default:
                System.out.println("Invalid choice");
                break;
        }
    }
}

```

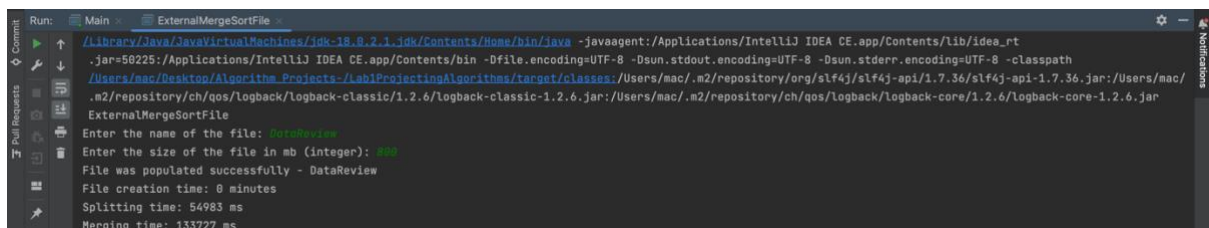
## ВИСНОВОК

У цій роботі ми розробили алгоритм прямого злиття (External Merge Sort), який є ефективним у сортуванні великих наборів даних, коли обсяг інформації перевищує доступну оперативну пам'ять. Алгоритм включає 3 головних етапи:

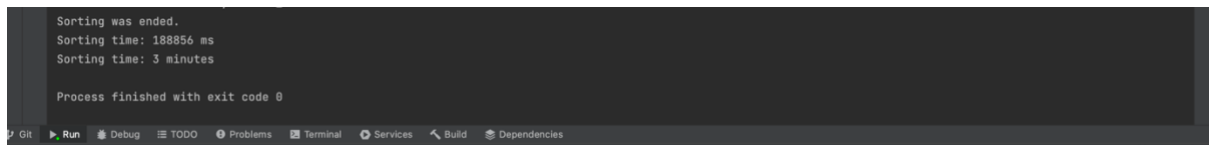
1. Розділення на чанки
2. Сортування кожного чанку
3. Злиття чанків в один файл

## РЕЗУЛЬТАТИ

### ExternalMergeSortFile

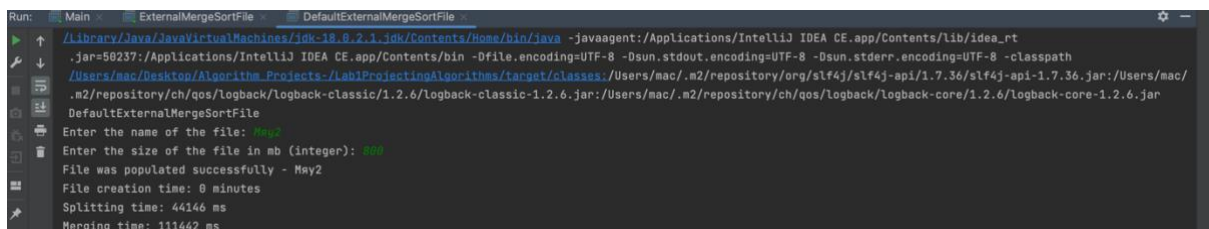


```
Run: Main ExternalMergeSortFile
/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50225:/Applications/IntelliJ IDEA CE.app/Contents/bin -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath /Users/mac/Desktop/Algorithm-Projects/1stProjectingAlgorithms/target/classes:/Users/mac/.m2/repository/org/slf4j/slf4j-api/1.7.36/slf4j-api-1.7.36.jar:/Users/mac/.m2/repository/ch/qos/logback/logback-classic/1.2.6/logback-classic-1.2.6.jar:/Users/mac/.m2/repository/ch/qos/logback/logback-core/1.2.6/logback-core-1.2.6.jar
ExternalMergeSortFile
Enter the name of the file: DataReview
Enter the size of the file in mb (integer): 800
File was populated successfully - DataReview
File creation time: 0 minutes
Splitting time: 54983 ms
Merging time: 133727 ms
```

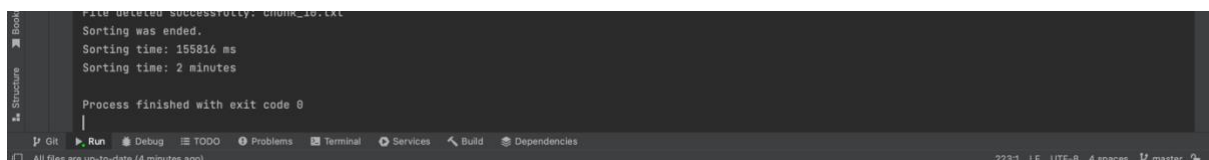


```
Sorting was ended.
Sorting time: 188856 ms
Sorting time: 3 minutes
Process finished with exit code 0
```

### DefaultExternalMergeSortFile



```
Run: Main ExternalMergeSortFile DefaultExternalMergeSortFile
/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50237:/Applications/IntelliJ IDEA CE.app/Contents/bin -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath /Users/mac/Desktop/Algorithm-Projects/1stProjectingAlgorithms/target/classes:/Users/mac/.m2/repository/org/slf4j/slf4j-api/1.7.36/slf4j-api-1.7.36.jar:/Users/mac/.m2/repository/ch/qos/logback/logback-classic/1.2.6/logback-classic-1.2.6.jar:/Users/mac/.m2/repository/ch/qos/logback/logback-core/1.2.6/logback-core-1.2.6.jar
DefaultExternalMergeSortFile
Enter the name of the file: May2
Enter the size of the file in mb (integer): 800
File was populated successfully - May2
File creation time: 0 minutes
Splitting time: 44146 ms
Merging time: 111442 ms
```



```
File deleted successfully: chunk_10.txt
Sorting was ended.
Sorting time: 155816 ms
Sorting time: 2 minutes
Process finished with exit code 0
```

Отже, ефективнішим виявився алгоритм DefaultExternalMergeSortFile, тому що файл на 2.3 гб був просортований за 2 хв, коли ExternalMergeSortFile був просортований за 3 хв. Тож, за 1 хв DefaultExternalMergeSortFile просортував 1, 15 гб, а ExternalMergeSortFile

за 1 хв – 0.77. Також усі інші функції такі як split, merge виконуються в першому варіанті (DefaultExternalMergeSortFile) швидше, це відбувається через те, що даний файл містить оптимізований спосіб сортування частин і об'єднання їх – такий як черга з пріоритетами.

### Відмінності між реалізаціями **DefaultExternalMergeSortFile** and **ExternalMergeSortFile**:

Обидва наведені вами коди є реалізаціями алгоритму сортування методом прямого злиття (merge sort). Однак вони мають деякі відмінності в логіці та структурі коду:

#### **Розділення та сортування чанків:**

У першому коді (ExternalMergeSortFile) розділення та сортування чанків відбувається в методі splitInputFile, де кожен чанк сортується окремо перед записом у файл. У другому коді (DefaultExternalMergeSortFile) це відбувається в методі splitInputFile, але сортування відбувається тільки перед записом у чанк, і пізніше вони об'єднуються в одну велику відсортовану послідовність під час злиття.

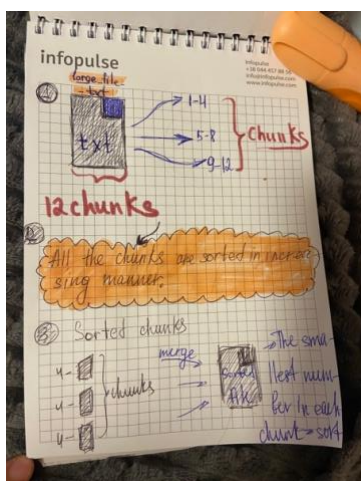
#### **Злиття великих чанків:**

У першому коді, після розділення та сортування чанків, вони об'єднуються в один великий відсортований файл в методі mergeSortedChunks за допомогою мінімального кучера.

У другому коді, чанки зливаються на льоту під час обробки, і вони не об'єднуються в один великий файл. Замість цього, використовується пріоритетна черга для зберігання мінімальних значень з різних чанків.

#### **Правило вибору мінімального елемента:**

У першому коді, мінімальний елемент знаходиться за допомогою методу findMinIndex, який проходиться по всіх елементах, щоб знайти найменший. У другому коді, мінімальні елементи знаходяться за допомогою пріоритетної черги, що дозволяє отримувати мінімальний елемент за  $O(\log n)$  часу.





## 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%.