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-13 Кисельов Микита* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(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**
12	Багатофазне сортування

3 ВИКОНАННЯ

3.1 Псевдокод алгоритму

БАГАТОФАЗНЕ СОРТУВАННЯ

1. ПОЧАТОК
2. СТВОРИТИ N ДОПОМІЖНИХ ФАЙЛІВ
3. РОЗДІЛИТИ ПОЧАТКОВИЙ ФАЙЛ ПО СЕРІЯМ ЗА ПРАВИЛОМ ФІБОНАЧІ У ДОПОМІЖНІ ФАЙЛИ
4. ПОКИ КІЛЬКІСТЬ ЗЛИТИХ СЕРІЙ ЗА РІВЕНЬ БІЛЬШЕ 0 ПОВТОРИТИ
   4.1. ЗЛИТИ ФАЙЛИ ЗА ПРАВИЛОМ РОЗПОДІЛУ ФІБОНАЧІ ДАНОГО РІВНЮ
   4.2. ПОНИЗИТИ РІВЕНЬ ПРАВИЛА РОЗПОДІЛУ ФІБОНАЧІ НА ОДИН ПОРЯДОК
5. КІНЕЦЬ

РОЗПОДІЛ ПОЧАТКОВИХ ФАЙЛІВ ЗА ПРАВИЛОМ ФІБОНАЧІ

1. ПОЧАТОК
2. ДЕКЛАРУВАННЯ ЗМІННИХ
   2.1. СЕРІЙ_ЗА_РІВЕНЬ = 1
   2.2. МАСИВ_РІВНЯ[0] = 1
   2.3. ФАЙЛ_ВИВОДУ = ДЛИНА ДОПОМІЖНИХ ФАЙЛІВ – 1
3. ПОКИ ПРОЧИТАНІ ДАНІ МЕНШІ ЗА ЗАГАЛЬНІ ПОВТОРИТИ
   3.1. ПОВТОРИТИ ВІД i = 0 ДО N
      3.1.1. ПОКИ ПРОЧИТАНІ ДАНІ МЕНШІ ЗА ЗАГАЛЬНІ ТА ОСТАННІЙ ЕЛЕМЕНТ СЕРІЇ[i] МЕНШЕ НАСТУПНОГО ЕЛЕМЕНТА ПОВТОРИТИ
         3.1.1.1. ЗАПИСАТИ НАСТУПНИЙ ЕЛЕМЕНТ У ПРАВИЛЬНО ВИЗНАЧЕНУ СЕРІЮ

3.1.2. ЗАПИСАТИ НАСТУПНИЙ ЕЛЕМЕНТ У ПРАВИЛЬНО ВИЗНАЧЕНУ СЕРІЮ

3.1.3. КІЛЬКІСТЬ ФІКТИВНИХ СЕРІЙ += 1

3.1.4. ЗАПИСАНА_ПОСЛІДОВНІСТЬ[i] += 1

3.2. ПОСТАВИТИ НАСТУПНИЙ РІВЕНЬ РОЗПОДІЛУ ЗА ПРАВИЛОМ ФІБОНАЧІ

4. ПОСТАВИТИ ПОПЕРЕДНІЙ РІВЕНЬ РОЗПОДІЛУ ЗА ПРАВИЛОМ ФІБОНАЧІ
5. ЗАПИСАТИ У МАСИВ ФІКТИВНИХ СЕРІЙ ЇХ ЗАЛИШОК ПІСЛЯ РОЗПОДІЛУ
6. КІНЕЦЬ

ЗЛИТТЯ ФАЙЛІВ ЗА ПРАВИЛОМ ФІБОНАЧІ

1. ПОЧАТОК
2. ЗАПОВНИТИ ПРИОРИТЕТНУ ЧЕРГУ ДОСТУПНИМИ ДЛЯ ЧИТАННЯ ЕЛЕМЕНТАМИ ІЗ ІНДЕКСОМ ЇХ ФАЙЛА
3. ПОКИ ЧЕРГА НЕ СТАЛА ПУСТОЮ МІНІМАЛЬНИМ РЕАЛЬНИМ СЕРІЯМ У ФАЙЛІ РАЗІВ ПОВТОРИТИ

3.1. ЯКЩО ЕЛЕМЕНТ НЕ ПУСТИЙ

3.1.1. ЗАПИСАТИ У ФАЙЛА ВИВОДУ ЕЛЕМЕНТ

3.1.2. ЯКЩО МОЖНА ПРОЧИТАТИ ЕЛЕМЕНТ

3.1.2.1. ДОДАТИ ЕЛЕМЕНТ У ПРИОРИТЕТНУ ЧЕРГУ

3.1.3. ЯКЩО ЧЕРГА ПУСТА

3.1.3.1. КІЛЬКІСТЬ_РАЗІВ_ПУСТОЇ_ЧЕРГИ += 1

3.1.3.2. ПОВТОРИТИ ДЛЯ i ВІД 0 ДО N

3.1.3.2.1. ЯКЩО i ЕЛЕМЕНТ НЕ ПУСТИЙ

3.1.3.2.1.1. ДОДАТИ ЕЛЕМЕНТ У ЧЕРГУ

3.1.3.2.1.2. ОСТАННІЙ_ЕЛЕМЕНТ_ДАНОЇ_СЕРІЇ[i] = i ЕЛЕМЕНТ

3.1.3.3. ЗАПОВНИТИ ПРИОРИТЕТНУ ЧЕРГУ ДОСТУПНИМИ ЕЛЕМЕНТАМИ

4. КІНЕЦЬ

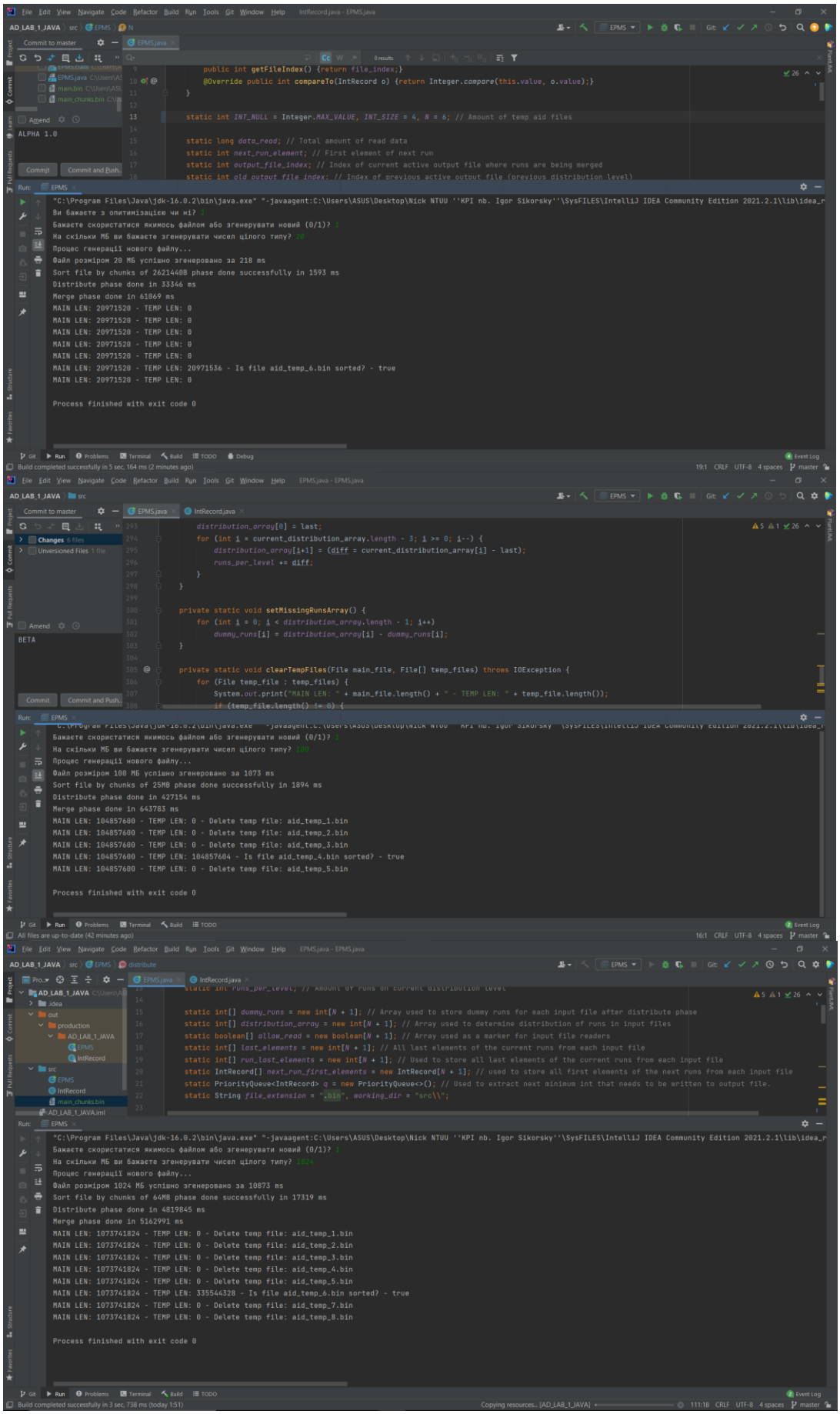
3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

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

public class EPMS  {
    public record IntRecord(int value, int file_index)
implements Comparable<IntRecord> {
        public int getValue() {return value;}
        public int getFileIndex() {return file_index;}
        @Override public int compareTo(IntRecord o) {return
Integer.compare(this.value, o.value);}
    }

    static int INT_NULL = Integer.MAX_VALUE, INT_SIZE = 4, N
= 5; // Amount of temp aid files

    static long data_read; // Total amount of read data
    static int next_run_element; // First element of next run
    static int output_file_index; // Index of current active
output file where runs are being merged
    static int old_output_file_index; // Index of previous
active output file (previous distribution level)
    static int runs_per_level; // Amount of runs on current
distribution level

    static int[] dummy_runs = new int[N + 1]; // Array used
to store dummy runs for each input file after distribute
phase
    static int[] distribution_array = new int[N + 1]; //
Array used to determine distribution of runs in input files
    static boolean[] allow_read = new boolean[N + 1]; //
Array used as a marker for input file readers
    static int[] last_elements = new int[N + 1]; // All last
elements of the current runs from each input file
    static int[] run_last_elements = new int[N + 1]; // Used
to store all last elements of the current runs from each
input file
    static IntRecord[] next_run_first_elements = new
IntRecord[N + 1]; // used to store all first elements of the
next runs from each input file
    static PriorityQueue<IntRecord> q = new
PriorityQueue<>(); // Used to extract next minimum int that
needs to be written to output file.
    static byte[] writeBuffer = new byte[4];

    static long mainFileLength;

    public static void main(String[] args) throws IOException
{
        System.out.print("Ви бажаєте з опитимізацією чи ні?
");
        boolean isOptimized = Integer.parseInt(new
BufferedReader(new InputStreamReader(System.in)).readLine())
> 0;
        File file = initFile( "main.bin");
        mainFileLength = file.length();
        initSort(isOptimized ? sortFileByChunks(file,
(int)Math.min(file.length() >> 3, 1 << 24)) : file);
    }

    private static File sortFileByChunks(File target_file,
int bLen) throws IOException {
        File result_file = new
File(target_file.getPath().replaceFirst(".bin", "") +
"_chunks.bin");
        DataInputStream dis = new DataInputStream(new
FileInputStream(target_file));
        DataOutputStream dos = new DataOutputStream(new
FileOutputStream(result_file));
        byte[] b = new byte[bLen];
        int[] int_buff = new int[bLen >> 2];
        long start = System.currentTimeMillis();
        while (dis.read(b) >= INT_SIZE) {
            for (int i = 0; i < int_buff.length; i +=
INT_SIZE)
                int_buff[i >> 2] = b[i] << 24 | (b[i + 1] &
0xFF) << 16 | (b[i + 2] & 0xFF) << 8 | (b[i + 3] & 0xFF);
            Arrays.sort(int_buff);
            dos.write(toByte(int_buff));
        }
        System.out.println("Sort file by chunks of " + bLen +
"B phase done successfully in " +
(System.currentTimeMillis()-start) + " ms");
        return result_file;
    }

    private static byte[] toByte(int[] d) throws IOException
{
        ByteArrayOutputStream bos = new
ByteArrayOutputStream(d.length << 2);
        DataOutputStream dos = new DataOutputStream(bos);
        for (int i : d) dos.writeInt(i);
        return bos.toByteArray();
    }

    private static void initSort(File main_file) throws
IOException {
        DataInputStream[] run_files_dis = new
DataInputStream[N + 1];
        File[] working_files = new File[N + 1];
        for (int i = 0; i < working_files.length; i++)
            working_files[i] = new File("aid_temp_" + (i+1) +
".bin");
        distribute(N, working_files, new DataInputStream(new
FileInputStream(main_file)));
        long start = System.currentTimeMillis();
            int min_dummy_values = getMinDummyValue();
            initMergeProcedure(min_dummy_values);
            DataOutputStream dos = new DataOutputStream(new
FileOutputStream(working_files[output_file_index]));
            for (int i = 0; i < run_files_dis.length - 1;
i++)
                run_files_dis[i] = new DataInputStream(new
FileInputStream(working_files[i]));
            while (runs_per_level >= 0) {
                last_elements[output_file_index] = INT_NULL;
                merge(distribution_array[getMinFileIndex()] -
min_dummy_values, run_files_dis, dos);
                setPreviousRunDistributionLevel();
                output_file_index = (output_file_index > 0 ?
output_file_index : distribution_array.length) - 1;
                resetAllowReadArray();
                min_dummy_values = getMinDummyValue();
                dos = new DataOutputStream(new
FileOutputStream(working_files[output_file_index]));
                run_files_dis[old_output_file_index] = new
DataInputStream(new
FileInputStream(working_files[old_output_file_index]));
            }
        System.out.println("Merge phase done in " +
(System.currentTimeMillis() - start) + " ms");
        dos.close();
        closeAll(run_files_dis);
        outAidFiles(main_file, working_files);
    }

    private static void distribute(int temp_files, File[]
working_files, DataInputStream main_file_dis) throws
IOException {
        long start = System.currentTimeMillis();
        runs_per_level = 1;
        distribution_array[0] = 1;
        output_file_index = working_files.length - 1;
        int[] write_sentinel = new int[temp_files];
        DataOutputStream[] run_files_dos = new
DataOutputStream[temp_files];
        for (int i=0; i < temp_files; i++)
            run_files_dos[i] = new DataOutputStream(new
FileOutputStream(working_files[i]));
        while (data_read < mainFileLength) {
            for (int i=0; i < temp_files; i++)
                while (write_sentinel[i] !=
distribution_array[i]) {
                    while (data_read < mainFileLength &&
next_run_element != INT_NULL && run_last_elements[i] <=
next_run_element)
                        writeNextIntRun(main_file_dis,
run_files_dos[i], i);
                    writeNextIntRun(main_file_dis,
run_files_dos[i], i);
                    dummy_runs[i]++;
                    write_sentinel[i]++;
                }
```

```
            setNextDistributionLevel();
        }
        setPreviousRunDistributionLevel();

        // set missing runs array
        for (int i = 0; i < distribution_array.length - 1;
i++)
            dummy_runs[i] = distribution_array[i] -
dummy_runs[i];

        System.out.println("Distribute phase done in " +
(System.currentTimeMillis() - start) + " ms");
    }

    private static void initMergeProcedure(int min_dummy) {
        for (int i=0; i < dummy_runs.length - 1; i++)
dummy_runs[i] -= min_dummy;
        dummy_runs[output_file_index] += min_dummy;
        resetAllowReadArray();
    }

    private static void merge(int min_file_values,
DataInputStream[] run_files_dis, DataOutputStream writer)
throws IOException {
        int num, min_file, heap_empty = 0;
        IntRecord record;
        populateHeap(run_files_dis);
        while (heap_empty != min_file_values) {
            if ((record = q.poll()) == null) return;
            writer.writeInt(record.getValue());
            min_file = record.getFileIndex();
            if (allow_read[min_file])
                if ((num = readInt(run_files_dis[min_file],
min_file)) != INT_NULL)
                    q.add(new IntRecord(num, min_file));
            if (q.size() == 0) {
                heap_empty++;
                for (int i = 0; i <
next_run_first_elements.length; i++) {
                    if (next_run_first_elements[i] == null)
break;
                    q.add(new
IntRecord(next_run_first_elements[i].getValue(), i));
                    last_elements[i] =
next_run_first_elements[i].getValue();
                }
                populateHeap(run_files_dis);
                resetAllowReadArray();
            }
        }
    }
    private static void populateHeap(DataInputStream[]
run_files_dis) throws IOException {
        for (int i = 0; i < run_files_dis.length; i++)
            if (dummy_runs[i] == 0) {
                if (allow_read[i]) q.add(new
IntRecord(readInt(run_files_dis[i], i), i));
            } else dummy_runs[i]--;
    }
    private static int readInt(DataInputStream file_dis, int
file_index) throws IOException {
        if (file_dis.read(writeBuffer) < INT_SIZE) return
INT_NULL;
        int current_int = buffToInt();
        if (last_elements[file_index] != INT_NULL)
            if (current_int < last_elements[file_index]) {
                next_run_first_elements[file_index] = new
IntRecord(current_int, file_index);
                allow_read[file_index] = false;
                return INT_NULL;
            }
        return last_elements[file_index] = current_int;
    }
    private static void resetAllowReadArray() {
        Arrays.fill(allow_read, true);
        allow_read[output_file_index] = false;
    }
    private static int getMinDummyValue() {
        int min = dummy_runs[0];
        for (int i = 1; i < dummy_runs.length; i++)
            if  (dummy_runs[i] < min)
                if (i != output_file_index) min =
dummy_runs[i];
        return min;
    }
    private static int getMinFileIndex() {
        int min_file_index = 0, min = distribution_array[0];
        for (int i = 1; i < distribution_array.length; i++)
            if (distribution_array[i] != 0)
                if (distribution_array[i] < min)
min_file_index = i;
        return min_file_index;
    }
    private static void writeNextIntRun(DataInputStream
main_file_dis, DataOutputStream run_file_dos, int file_index)
throws IOException {
        if (data_read >= mainFileLength)
{dummy_runs[file_index]--; return;}
        if (next_run_element != INT_NULL) {
            run_file_dos.writeInt(next_run_element);
            data_read += INT_SIZE;
        }
        int min_value = Integer.MIN_VALUE;
        if (main_file_dis.read(writeBuffer) < INT_SIZE)
return;
        int current_int = buffToInt();
        while(current_int !=  INT_NULL) {
            if (current_int >= min_value) {
                run_file_dos.writeInt(current_int);
                data_read += INT_SIZE;
                min_value = current_int;
                if (main_file_dis.read(writeBuffer) <
INT_SIZE) break;
                current_int = buffToInt();
            } else {
                next_run_element = current_int;
                run_last_elements[file_index] = min_value;
                break;
            }
        }
    }

    private static int buffToInt() {
        return writeBuffer[0] << 24 | (writeBuffer[1] & 0xFF)
<< 16 | (writeBuffer[2] & 0xFF) << 8 | (writeBuffer[3] &
0xFF);
    }
    private static void setNextDistributionLevel() {
        runs_per_level = 0;
        int[] clone = distribution_array.clone();
        for (int i = 0; i < clone.length - 1; runs_per_level
+= distribution_array[++i])
            distribution_array[i] = clone[0] + clone[i+1];
    }
    private static void setPreviousRunDistributionLevel() {
        int[] clone = distribution_array.clone();
        old_output_file_index = output_file_index;
        distribution_array[0] = runs_per_level =
clone[clone.length - 2];
        for (int diff, i = clone.length - 3; i >= 0; i--,
runs_per_level += diff)
            distribution_array[i+1] = diff = clone[i] -
clone[clone.length - 2];
    }
    private static void outAidFiles(File main_file, File[]
temp_files) throws IOException {
        for (File temp_file : temp_files) {
            System.out.print("MAIN LEN: " +
main_file.length() + " - TEMP LEN: " + temp_file.length());
            System.out.println(temp_file.length() != 0 ? " -
Is file " + temp_file.getName() + " sorted? - " +
isFileSorted(temp_file) : "");
        }
    }
    private static boolean isFileSorted(File temp_file)
throws IOException {
        DataInputStream dis = new DataInputStream(new
FileInputStream(temp_file));
        if (dis.read(writeBuffer) < INT_SIZE) return false;
        int first = buffToInt();
        if (dis.skip(temp_file.length() - 8) <= 0) return
false;
        if (dis.read(writeBuffer) < INT_SIZE) return false;
        dis.close();
        return first <= buffToInt();
    }
    private static void closeAll(Closeable[] c) throws
IOException {for (Closeable o: c) o.close();}

    public static File initFile(String path) throws
IOException {
        System.out.print("Бажаєте скористатися якимось файлом
або згенерувати новий (0/1)? ");
        BufferedReader console = new BufferedReader(new
InputStreamReader(System.in));
        File data;
        if (Integer.parseInt(console.readLine()) > 0) {
            System.out.print("На скільки МБ ви бажаєте
згенерувати чисел цілого типу? ");
            int mb = Integer.parseInt(console.readLine());
            DataOutputStream dos = new DataOutputStream(new
FileOutputStream(data = new File(path)));
            Random random = new Random();
            System.out.println("Процес генерації нового
файлу...");
            long start = System.currentTimeMillis();
            if (mb <= 1 << 8) {
                int[] tmp = new int[mb << 18];
                for (int i = 0; i < mb << 18; i++) tmp[i] =
random.nextInt(INT_NULL);
                dos.write(toByte(tmp));
            } else {
                int[] tmp = new int[1 << 8];
                for (int i = 0; i < mb << 10; i++) {
                    for (int j = 0; j < 1 << 8; j++) tmp[j] =
random.nextInt(INT_NULL);
                    dos.write(toByte(tmp));
                }
            }
            System.out.println("Файл розміром " + mb + " МБ
успішно згенеровано за " + (System.currentTimeMillis()-start)
+ " ms");
            dos.close();
        } else {
            System.out.print("Вкажіть шлях до файлу: ");
            data = new File(console.readLine());
        }
        console.close();
        return data;
    }
}
```

3.2.2 Приклад роботи



3.2.3 Часові характеристики алгоритму

1. Неоптимізований алгоритм (допоміжні файли – 3):
   1.1. 20 МБ – 441 секунда;
   1.2. 1 ГБ – 1;
2. Оптимізований алгоритм (допоміжні файли – 7):
   2.1. 20 МБ – 72 секунди;
   2.2. 1 ГБ – 3,5 години.

ВИСНОВОК

При виконанні даної лабораторної роботи я реалізував один із алгоритмів зовнішнього сортування, а саме багатофазне сортування. Я розробив програмну реалізацію даного алгоритму за допомогою мови програмування Java та відсортував файл розміром 20971520Б типу даних Integer, що становить 20 мегабайт. На сортування було витрачено 441 секунд. Параметри сортування були такими: кількість допоміжних файлів – 3. Також я створив модифіковану версію алгоритму. За її допомогою цей же файл було відсортовано за 72 секунди при наступних параметрах: кількість допоміжних файлів – 7, розмір однієї серії – 5МБ.

За допомогою використання модифікованої версії програми був відсортований файл розміром 1ГБ. Сортування тривало 40 хвилин при таких параметрах: кількість допоміжних файлів – 7, кількість мегабайт у одній серії – 512. Таким чином я досяг швидкості сортування 2,5 години на файл, розмір якого у 2 рази більший, ніж обсяг оперативної пам’яті ПК, тобто 4 ГБ (за рахунком того, що віртуальна Java машина JVM обмежує кількість доступної оперативної пам’яті до 2ГБ).

Модифікована версія програми відрізняється від звичайної записом та читанням з файлів. Звичайна версія зчитувала по одному числу з файла, а потім записувала це число до іншого файлу. Модифікація полягала у тому, аби зчитати масив байтів та перетворити його на масив цілих чисел. Подальші дії вже виконувалися саме з числами у масиві, а не самому файлі. Те ж саме стосується і запису – замість того, щоб записувати по одному числу до файлу, я записував ці числа у масив цілих чисел, перетворював даний масив у масив байтів і вже потім записував всі байти одразу до файлу. Таким чином мені вдалося зменшити кількість звертань до файлів, що пришвидшило роботу програми.

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%.