

**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”**

**Варіант 20**

**Виконав(ла)**

III-15, Плугатирьов Д.В.  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

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

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>6</b>
3.1	ПСЕВДОКОД АЛГОРИТМУ.....	6
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ .....	8
3.2.1	<i>Вихідний код.....</i>	8
	<b>ВИСНОВОК .....</b>	<b>22</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>23</b>

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

## 3.1 Псевдокод алгоритму

**DistributeFileOnSeries (S)**

S розмір створюваних відрізків

CurrentFile = A

while кінець другого файлу не досягнутий do

    read S записів із вхідного файлу

    sort S записів

    if CurrentFile = A then

        CurrentFile = B

    else

        CurrentFile = A

    end if

end while

**PolyPhaseMerge(S)**

S розмір створюваних відрізків

Size = S

Input1 = A

Input2 = B

CurrentOutput = C

while not done do

    while відрізки не закінчились do

        злити серію довжини Size з файла Input1

        із серією довжини Size з файла Input2

        записавши результат в CurrentOutput

```
    if (CurrentOutput = A) then
        CurrentOutput = B
    elseif (CurrentOutput = B) then
        CurrentOutput = A
    elseif (CurrentOutput = C) then
        CurrentOutput = D
    elseif (CurrentOutput = D) then
        CurrentOutput = C
    end if
end while
```

```
Size = Size * 2
```

```
if (Input1 = A) then
    Input1 = C
    Input2 = D
    CurrentOutput = A
else
    Input1 = A
    Input2 = B
    CurrentOutput = C
end if
end while
```

## 3.2 Програмна реалізація алгоритму

### 3.2.1 Вихідний код

```
public class OutputHandler
{
    public static void PrintEnumerable(IEnumerable enumerable)
    {
        foreach (var item in enumerable)
        {
            System.Console.Write($"{item} ");
        }
    }
}
```

```
public class FileHandler
{
    public static int Mb = 1024 * 1024;
    public static string AfilePath = "A.bin";
    public static string BfilePath = "B.bin";
    public static string CfilePath = "C.bin";
    public static string DfilePath = "D.bin";
    public static string initialFilePath = "initial.bin";
    public static FileInfo initialFileInfo = new
    FileInfo(initialFilePath);

    public void WriteArrayToFile(BinaryWriter bw, long[] arr)
    {
        foreach (long item in arr)
        {
            bw.Write(item);
        }
    }

    public void CreateFileWithRandomNumbers(int megabytesCount,
    string fileName)
    {
        long fileSize = megabytesCount * Mb;

        using (BinaryWriter fs = new
    BinaryWriter(File.Create(fileName)))
        {
            Random rd = new Random();
        }
    }
}
```



```

        for (long i = 0; i < fileSize; i += sizeof(long))
        {
            fs.Write((long)rd.Next());
        }
    }

    public void CreateFileWithRandomNumbers(string fileName, int
elementsCount)
    {
        using (BinaryWriter fs = new BinaryWriter(new
FileStream(fileName, FileMode.Create)))
        {
            Random rd = new Random();

            for (int i = 0; i < elementsCount; i++)
            {
                fs.Write((long)rd.Next(1, 1000));
            }
        }

        public long[] GetFileContent(string fileName)
        {
            List<long> result = new List<long>();

            using (BinaryReader br = new BinaryReader(File.Open(fileName,
FileMode.Open)))
            {
                while (br.BaseStream.Position != br.BaseStream.Length)
                {
                    result.Add(br.ReadInt64());
                }
            }

            return result.ToArray();
        }
    }
}

```

```

public class SortingHandler
{
    private FileHandler _fileHandler;
}

```

```

public SortingHandler(FileHandler fileHandler)
{
    _fileHandler = fileHandler;
}

public void QuickSort(long[] a, int lo, int hi)
{
    if (lo >= hi)
    {
        return;
    }

    int lt = lo; int i = lo + 1; int gt = hi;
    long t = a[lo];

    while (i <= gt)
    {
        int cmp = a[i].CompareTo(t);
        if (cmp < 0)
        {
            Exchange(a, i++, lt++);
        }
        else if (cmp > 0)
        {
            Exchange(a, i, gt--);
        }
        else
        {
            i++;
        }
    }

    QuickSort(a, lo, lt - 1);
    QuickSort(a, gt + 1, hi);
}

public void Exchange(long[] a, int i, int j)
{
    long t = a[i];
    a[i] = a[j];
    a[j] = t;
}

```

```

    public void QuickSort(long[] a)
    {
        for (int i = 0; i < a.Length - 1; i++)
        {
            if (a[i] > a[i + 1])
            {
                QuickSort(a, 0, a.Length - 1);
                break;
            }
        }
    }

    /// <param name="sectionSize">count of Mb (numbers count in DEBUG
mode)</param>
    public void DistributeFileOnSeries(int sectionSize)
    {
        int elementsCount;

        if (Program.DEBUG)
        {
            elementsCount = sectionSize;
            sectionSize *= sizeof(long);
        }
        else
        {
            sectionSize *= FileHandler.Mb;
            elementsCount = sectionSize / sizeof(long);
        }

        using FileStream Afile = new
FileStream(FileHandler.AfilePath, FileMode.Create);
        using FileStream Bfile = new
FileStream(FileHandler.BfilePath, FileMode.Create);
        BinaryWriter currentFile = new BinaryWriter(Afile);

        using (MemoryMappedFile initialFile = MemoryMappedFile
.CreateFromFile(FileHandler.initialFilePath,
FileMode.Open))
        {
            using (MemoryMappedViewAccessor accessor =
initialFile.CreateViewAccessor())
            {
                long fileLength = FileHandler.initialFileInfo.Length;

```

```

        for (long i = 0; i < fileLength; i +=
(long)sectionSize)
        {
            long[] section = new long[elementsCount];
            accessor.ReadArray<long>(i, section, 0,
elementsCount);

            QuickSort(section);
            _fileHandler.WriteArrayToFile(currentFile,
section);

            if (currentFile.BaseStream == Afile)
            {
                currentFile = new BinaryWriter(Bfile);
            }
            else
            {
                currentFile = new BinaryWriter(Afile);
            }
        }
    }
}

/// <param name="sectionSize">count of Mb (numbers count in DEBUG
mode)</param>
public void PolyPhaseMerge(int sectionSize)
{
    int sectionElementsCount;

    if (Program.DEBUG)
    {
        sectionElementsCount = sectionSize;
        sectionSize *= sizeof(long);
    }
    else
    {
        sectionSize *= FileHandler.Mb;
        sectionElementsCount = sectionSize / sizeof(long);
    }

    long serieSize = sectionSize;
    long inputFileSize = FileHandler.initialFileInfo.Length / 2;

```

```

        string input1Name = FileHandler.AfilePath;
        MemoryMappedFile input1Map =
MemoryMappedFile.CreateFromFile(input1Name);
        MemoryMappedViewAccessor input1Accessor =
input1Map.CreateViewAccessor();

        string input2Name = FileHandler.BfilePath;
        MemoryMappedFile input2Map =
MemoryMappedFile.CreateFromFile(input2Name);
        MemoryMappedViewAccessor input2Accessor =
input2Map.CreateViewAccessor();

        File.Create(FileHandler.CfilePath).Close();
        File.Create(FileHandler.DfilePath).Close();

        string currentOutputName = FileHandler.CfilePath;
        BinaryWriter currentOutput = new
BinaryWriter(File.Open(currentOutputName, FileMode.Open));
        long fileLength = FileHandler.initialFileInfo.Length;

        while (serieSize != fileLength)
        {
            FileInfo input1Info = new FileInfo(input1Name);
            FileInfo input2Info = new FileInfo(input2Name);
            FileInfo currentOutputInfo = new
FileInfo(currentOutputName);

            long firstMapPosition = 0;
            long secondMapPosition = 0;
            long serieElementsCount = serieSize / sizeof(long);
            long[] input1Section = new long[sectionElementsCount];
            long[] input2Section = new long[sectionElementsCount];
            input1Accessor
                .ReadArray(firstMapPosition, input1Section, 0,
sectionElementsCount);
            input2Accessor
                .ReadArray(secondMapPosition, input2Section, 0,
sectionElementsCount);

            do
            {

```

```

        MergeSeries(sectionElementsCount, input1Section,
input2Section, currentOutput,
                    ref firstMapPosition, ref secondMapPosition,
sectionSize, serieSize,
                    input1Accessor, input2Accessor, inputFileSize);

        currentOutput.Close();

        ChangeOutput(ref currentOutputName, ref
currentOutput);
    }
    while ((firstMapPosition != inputFileSize) &&
(secondMapPosition != inputFileSize));

    serieSize *= 2;

    currentOutput.Close();
    input1Accessor.Dispose();
    input1Map.Dispose();
    input2Accessor.Dispose();
    input2Map.Dispose();

    File.Open(input1Name, FileMode.Truncate).Close();
    File.Open(input2Name, FileMode.Truncate).Close();

    if (serieSize != fileLength)
    {
        if (input1Name == FileHandler.AfilePath)
        {
            SwapFiles(ref input1Name, ref input1Map, ref
input1Accessor, ref input2Name,
                    ref input2Map, ref input2Accessor, ref
currentOutputName, ref currentOutput,
                    FileHandler.CfilePath, FileHandler.DfilePath,
FileHandler.AfilePath);
        }
        else
        {
            SwapFiles(ref input1Name, ref input1Map, ref
input1Accessor, ref input2Name,
                    ref input2Map, ref input2Accessor, ref
currentOutputName, ref currentOutput,

```

```

        FileHandler.AfilePath, FileHandler.BfilePath,
FileHandler.CfilePath);
    }
}

}

}

    public void ChangeOutput(ref string currentOutputName, ref
BinaryWriter currentOutput)
    {
        if (currentOutputName == FileHandler.AfilePath)
        {
            currentOutputName = FileHandler.BfilePath;
            currentOutput = new
BinaryWriter(File.Open(currentOutputName, FileMode.Append));
        }
        else if (currentOutputName == FileHandler.BfilePath)
        {
            currentOutputName = FileHandler.AfilePath;
            currentOutput = new
BinaryWriter(File.Open(currentOutputName, FileMode.Append));
        }
        else if (currentOutputName == FileHandler.CfilePath)
        {
            currentOutputName = FileHandler.DfilePath;
            currentOutput = new
BinaryWriter(File.Open(currentOutputName, FileMode.Append));
        }
        else if (currentOutputName == FileHandler.DfilePath)
        {
            currentOutputName = FileHandler.CfilePath;
            currentOutput = new
BinaryWriter(File.Open(currentOutputName, FileMode.Append));
        }
    }

    public void SwapFiles(ref string input1Name, ref MemoryMappedFile
input1Map,
        ref MemoryMappedViewAccessor input1Accessor, ref string
input2Name,
        ref MemoryMappedFile input2Map, ref MemoryMappedViewAccessor
input2Accessor,

```

```

        ref string currentOutputName, ref BinaryWriter currentOutput,
string newInput1Name,
        string newInput2Name, string newOutputName)
    {
        input1Name = newInput1Name;
        input1Map = MemoryMappedFile.CreateFromFile(input1Name);
        input1Accessor = input1Map.CreateViewAccessor();

        input2Name = newInput2Name;
        input2Map = MemoryMappedFile.CreateFromFile(input2Name);
        input2Accessor = input2Map.CreateViewAccessor();

        currentOutputName = newOutputName;
        currentOutput = new BinaryWriter(File.Open(currentOutputName,
FileMode.Open));
    }

    public void MergeSeries(int sectionElementsCount, long[]
input1Section, long[] input2Section,
        BinaryWriter currentOutput, ref long firstMapPosition, ref
long secondMapPosition,
        long sectionSize, long serieSize, MemoryMappedViewAccessor
input1Accessor,
        MemoryMappedViewAccessor input2Accessor, long inputFileSize)
    {
        long firstSectionIndex = 0;
        long secondSectionIndex = 0;
        long tempPosition1 = 0;
        long tempPosition2 = 0;

        do
        {
            if ((firstSectionIndex != sectionElementsCount)
                && (secondSectionIndex != sectionElementsCount))
            {
                if (input1Section[firstSectionIndex] <
input2Section[secondSectionIndex])
                {
                    currentOutput.Write(input1Section[firstSectionInd
ex]);

                    firstSectionIndex++;
                }
                else

```



```

        {
            currentOutput.Write(input2Section[secondSectionIndex]);
            secondSectionIndex++;
        }
    }
    else if (firstSectionIndex == sectionElementsCount)
    {
        firstSectionIndex = 0;

        ProcessSectionEnding(sectionElementsCount,
            ref firstMapPosition, sectionSize, serieSize,
            input1Accessor,
            input1Section, ref secondSectionIndex,
            currentOutput, input2Section,
            ref secondMapPosition, input2Accessor,
            ref tempPosition1, ref tempPosition2,
            inputFileSize);
    }
    else if (secondSectionIndex == sectionElementsCount)
    {
        secondSectionIndex = 0;

        ProcessSectionEnding(sectionElementsCount,
            ref secondMapPosition, sectionSize, serieSize,
            input2Accessor,
            input2Section, ref firstSectionIndex,
            currentOutput, input1Section,
            ref firstMapPosition, input1Accessor,
            ref tempPosition2, ref tempPosition1,
            inputFileSize);
    }
}

while ((tempPosition1 != serieSize) && (tempPosition2 !=
serieSize));
}

public void ProcessSectionEnding(int sectionElementsCount,
    ref long firstMapPosition, long sectionSize, long serieSize,
    MemoryMappedViewAccessor input1Accessor, long[]
input1Section,
    ref long secondSectionIndex, BinaryWriter currentOutput,
    long[] input2Section,

```

```

        ref long secondMapPosition, MemoryMappedViewAccessor
input2Accessor,
        ref long tempPosition1, ref long tempPosition2, long
inputFileSize)
    {
        tempPosition1 = firstMapPosition % serieSize;
        tempPosition2 = secondMapPosition % serieSize;

        if (tempPosition1 != serieSize)
        {
            tempPosition1 += sectionSize;
        }

        firstMapPosition += sectionSize;

        if (firstMapPosition != inputFileSize)
        {
            input1Accessor.ReadArray(firstMapPosition, input1Section,
0, sectionElementsCount);
        }

        if (tempPosition1 == serieSize)
        {
            do
            {
                for (long i = secondSectionIndex; i <
sectionElementsCount; i++)
                {
                    currentOutput.Write(input2Section[i]);
                }

                secondSectionIndex = 0;

                if (tempPosition2 != serieSize)
                {
                    tempPosition2 += sectionSize;
                }

                secondMapPosition += sectionSize;

                if (secondMapPosition != inputFileSize)
                {

```

```

        input2Accessor.ReadArray(secondMapPosition,
input2Section, 0,
        sectionElementsCount);
    }
}
while (tempPosition2 != serieSize);
}
}
}

```

```

class Program
{
    public static bool DEBUG = false;

    public static void Main(string[] args)
    {
        FileHandler fileHandler = new FileHandler();
        const int SectionSize = 512;

        if (DEBUG)
        {
            int initialElementsCount = 128;

            if (!File.Exists(FileHandler.initialFilePath)
                || ((FileHandler.initialFileInfo.Length /
sizeof(long)) != initialElementsCount))
            {
                fileHandler
                    .CreateFileWithRandomNumbers(FileHandler.initialF
ilePath, initialElementsCount);
            }

            System.Console.WriteLine("Initial file:");
            OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.initialFilePath));
            System.Console.WriteLine();
        }
        else
        {
            int megabytesCount = 1024 * 8;

            if (!File.Exists(FileHandler.initialFilePath)

```

```

        || (FileHandler.initialFileInfo.Length !=
(megabytesCount * FileHandler.Mb)))
    {
        fileHandler.CreateFileWithRandomNumbers(megabytesCount, FileHandler.initialFilePath);
    }
}

Stopwatch timer = new Stopwatch();
timer.Start();
SortingHandler sortingHandler = new
SortingHandler(fileHandler);

sortingHandler.DistributeFileOnSeries(SectionSize);

System.Console.WriteLine($"Distribution of the initial file
took: {timer.Elapsed}");

if (DEBUG)
{
    System.Console.WriteLine("A:");
    OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.AfilePath));
    System.Console.WriteLine(Environment.NewLine);

    System.Console.WriteLine("B:");
    OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.BfilePath));
    System.Console.WriteLine(Environment.NewLine);
}

sortingHandler.PolyPhaseMerge(SectionSize);

timer.Stop();
System.Console.WriteLine($"Sorting took: {timer.Elapsed}");

if (DEBUG)
{
    System.Console.WriteLine("C:");
    OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.CfilePath));
    System.Console.WriteLine(Environment.NewLine);
}

```

```

        System.Console.WriteLine("D:");
        OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.DfilePath));
        System.Console.WriteLine(Environment.NewLine);

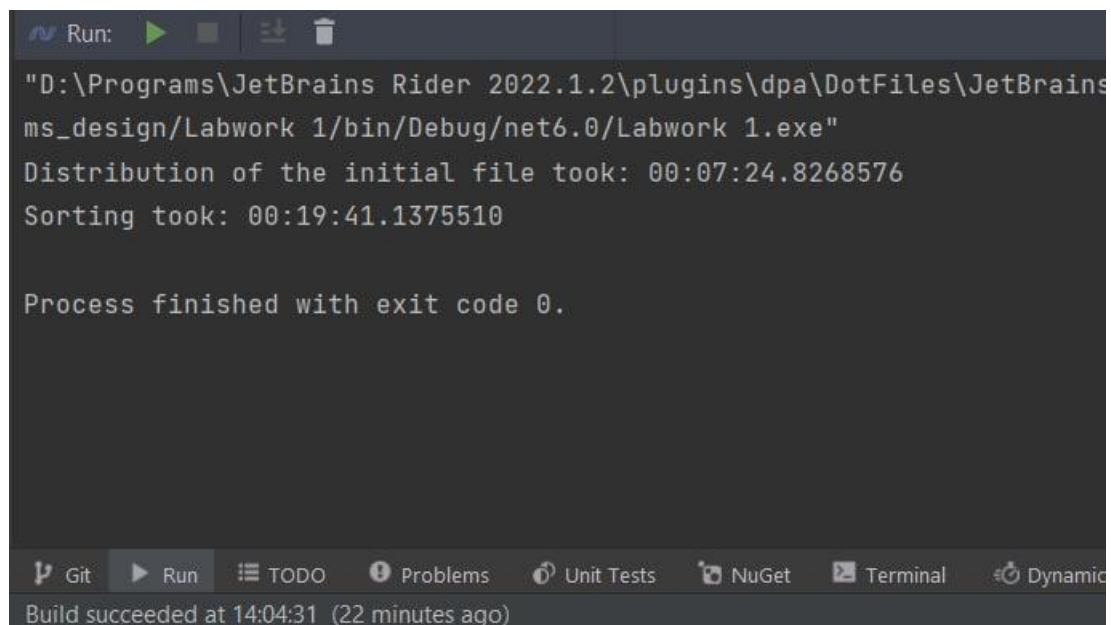
        System.Console.WriteLine("A:");
        OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.AfilePath));
        System.Console.WriteLine(Environment.NewLine);

        System.Console.WriteLine("B:");
        OutputHandler.PrintEnumerable(fileHandler.GetFileContent(
FileHandler.BfilePath));
        System.Console.WriteLine();
    }
}
}

```

### 3.2.2 Тестування алгоритму

В даному випадку, маючи обсяг оперативної пам'яті у 4 Гб та зазначивши серію розміром 512 Мб, було відсортовано 8 Гб-ний файл за такий проміжок часу:



```

Run: [Run] [Stop] [Debug] [Exit]
"D:\Programs\JetBrains Rider 2022.1.2\plugins\dpa\DotFiles\JetBrains
ms_design/Labwork 1/bin/Debug/net6.0/Labwork 1.exe"
Distribution of the initial file took: 00:07:24.8268576
Sorting took: 00:19:41.1375510

Process finished with exit code 0.

Git Run TODO Problems Unit Tests NuGet Terminal Dynamic
Build succeeded at 14:04:31 (22 minutes ago)

```

Рисунок 3.1 – приклад роботи програми

## ВИСНОВОК

При виконанні даної лабораторної роботи мені вдалося здобути навички стосовно роботи з файлами, котрі більші за об'єм оперативної пам'яті на ПК користувача. А саме, я реалізував алгоритм зовнішнього багатофазного сортування злиттям зв наведеним вище псевдокодом.

Тобто, я спочатку створюю файл розміром у степінь двійки (кількість Мб), потім розбиваю його на два файли А та Б таким чином: визначаю розмір секції та беру дані в залежності від нього з вхідного файлу поперемінно записуючи секцію у файли А та Б попередньо її відсортувавши алгоритмом Quicksort.

Кожен з файлів А і Б містить деяку послідовність відсортованих відрізків, проте, як і у випадку сортування злиттям, ми нічого не можемо сказати про порядок записів у двох різних відрізках. Процес злиття буде аналогічним до функції MergeLists, але тепер замість того, щоб переписувати записи у новий масив, ми будемо записувати їх в новий файл. Тому ми починаємо з читання половинок перших відрізків із файлів А і Б. Читаємо ми лише по половині відрізків, бо ми вже зрозуміли, що в пам'яті може розміщуватися одночасно лише  $S$  записів, а нам потрібні записи з обох файлів. Будемо тепер зливати ці половинки відрізків в один відрізок файлу С. Після того, як одна з половинок закінчиться, ми прочитаємо другу половинку з того ж файлу. Коли обробка одного з відрізків буде завершена, кінець другого відрізка буде переписаний в файл С. Після того, як злиття перших двох відрізків із файлів А і Б буде завершено, наступні два відрізки зливаються в файл Д. Цей процес злиття відрізків продовжується з поперемінним записом злитих відрізків у файли С і Д. По завершенню ми отримуємо два файли, розбиті на відсортовані відрізки довжини  $2S$ . Тоді процес повторюється, до того ж відрізки зчитуються з файлів С і Д, а злиті відрізки довжини  $4S$  записуються в файли А і Б. Ясно, що в кінці-кінців відрізки зіллються в один відсортований список в одному з файлів.

Після написання алгоритму, я виміряв час, за який він сортує файл певного розміру та він вийшов прийнятним.

## 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%.