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 Жмайло Д. О.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О. О.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 ЗАВДАННЯ

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

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

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

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

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

Таблиця 2.1 – Варіанти алгоритмів

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

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

3 ВИКОНАННЯ

Варіант 9

3.1 Псевдокод алгоритму

3.2 ЗАГАЛЬНИЙ АЛГОРИТМ

ПОЧАТОК

Ініціювати змінні

Ввести назву файлу

Ввести розмір файлу

Присвоїти numberOfElements := довжина файла у мегабайтах * 1024 * 1024 / 4

Згенерувати файл

Обрати тип сортування

Відсортувати файл у відповідності до обраного типу сортування

КІНЕЦЬ

3.3 АЛГОРИТМ ЗВИЧАЙНОГО ПРЯМОГО ЗЛИТТЯ

ПОЧАТОК

Цикл проходу по елементам файла, де i = 1, 2, 4, … кількість елементів

Розділити елементи

Відсортувати файли

КІНЕЦЬ

3.4 АЛГОРИТМ РОЗДІЛЕННЯ ЕЛЕМЕНТІВ

ПОЧАТОК

Відкрити на прочитання файл A, на запис файли B і C

Ініціалізація змінної count := 0

ПОКИ головний файл не прочитано до кінця ПОВТОРИТИ

ЯКЩО ((cnt / довжина поточної среії % 2) == 0) ТО:

Записати наступний елемент до файлу B

ІНАКШЕ

Записати наступний елемент до файлу C

count := count + 1

Закрити файли A, B і C

КІНЕЦЬ

3.5 АЛГОРИТМ СОРТУВАННЯ ЕЛЕМЕНТІВ

Відкрити на прочитання файл A, на запис файли B і C

Ініціалізація b, c, counterB, counterC цілочисельного типу

ПОКИ файл C не прочитано до кінця ПОВТОРИТИ

counterB := 0, CounterC := 0

b := прочитати наступний елемент файлу B

c := прочитати наступний елемент файлу C

ПОКИ (counterB < partSize && counterC < partSize) ПОВТОРИТИ

ЯКЩО b <= c ТО

Записати змінну b до файлу A

counterB := counterB + 1

ЯКЩО (counterB < partSize)

b := прочитати наступний елемент файлу B

ІНАКШЕ

ПОКИ (counterC < partSize) ПОВТОРИТИ

Записати змінну c до файлу A

counterC := counterC + 1

ЯКЩО (counterC < partSize && !Не кінець файлу C)

c := прочитати наступний елемент файлу C

ІНАКШЕ

Записати змінну C до файлу A

counterC := counterC + 1

ЯКЩО (counterC < partSize && !Не кінець файлу C)

c := прочитати наступний елемент файлу C

ІНАКШЕ

ПОКИ (counterB < partSize) ПОВТОРИТИ

Записати змінну b до файлу A

counterB := counterB + 1

ЯКЩО (counterB < partSize)

b := прочитати наступний елемент файлу B

ПОКИ файл B не прочитано до кінця ПОВТОРИТИ

Записати наступний елемент b з файлу B до файлу A

Закрити файли A, B і C

Видалити файли B і C

КІНЕЦЬ

3.6 Програмна реалізація алгоритму

3.6.1 Вихідний код

Program.cs

```
using System;
using System.Diagnostics;

namespace lab1
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Welcome to my program!");
            FileAssistant fileAssistant = new FileAssistant();
            InputAssistant inputAssistant = new InputAssistant();

            string filename = inputAssistant.GetFilename();
            int fileSize = inputAssistant.GetFileLength();
            long numberOfElements = fileSize * 1024 * 1024 / 4;

            Stopwatch sw = Stopwatch.StartNew();
            fileAssistant.GenerateFile(filename, fileSize);
            sw.Stop();
            Console.WriteLine($"File generated: {sw.ElapsedMilliseconds} ms" + "\n");

            SortAssistant sorter = inputAssistant.ChooseClasicOrOptimized();
            sw.Restart();
            sorter.Sort(filename, "tempB", "tempC", numberOfElements);
            sw.Stop();
            Console.WriteLine($"File sorted: {sw.ElapsedMilliseconds} ms");

            sw.Restart();
            if (fileAssistant.CheckFile(filename, 1000000))
                Console.WriteLine($"File sorted successfully! Check time is:
{sw.ElapsedMilliseconds} ms");
            else
                Console.WriteLine($"File sorted with an error! Check time is:
{sw.ElapsedMilliseconds} ms");
        }
    }
      }
```

FileAssistant.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lab1
{
    internal class FileAssistant
    {
        public void GenerateFile(string filename, int sizeOfFileInMb)
        {
            const int NUMBEROFELEMENTSINONEMB = 1024 * 1024 / sizeof(int);
            Random random = new Random();
            using (BinaryWriter writer = new BinaryWriter(new FileStream(filename,
FileMode.Create)))
            {
                for (int i = 0; i < sizeOfFileInMb; i++)
                {
                    for (int j = 0; j < NUMBEROFELEMENTSINONEMB; j++)
                        writer.Write(random.Next(int.MinValue, int.MaxValue));
                }
            }
        }

        protected bool StreamEnd(BinaryReader binReader)
        {
            if (binReader.BaseStream.Position == binReader.BaseStream.Length)
                return true;
            return false;
        }

        // Перевірити весь файл
        public bool CheckFile(string filename)
        {
            using (BinaryReader binaryReader = new BinaryReader(new
FileStream(filename, FileMode.Open)))
            {
                int prev = binaryReader.ReadInt32();
                while (!StreamEnd(binaryReader))
                {
                    int temp = binaryReader.ReadInt32();
                    if (temp >= prev)
                        prev = temp;
                    else
                        return false;
                }
                return true;
            }
        }

        // Перевірити файл частково
        public bool CheckFile(string filename, int count)
        {
            using (BinaryReader binReader = new BinaryReader(new FileStream(filename,
FileMode.Open)))
            {
                int prev = binReader.ReadInt32();
                while (!StreamEnd(binReader) && binReader.BaseStream.Position < count)
                {
                    int temp = binReader.ReadInt32();
                    if (temp >= prev)
                        prev = temp;
```

```
                    else
                        return false;
                }
                return true;
            }
        }
    }
}
```

MergeSort.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lab1
{
    internal class MergeSort : SortAssistant
    {

        protected void Split(string filepath, string tempB, string tempC, long
partSize)
        {
            BinaryReader binReader = new BinaryReader(new FileStream(filepath,
FileMode.OpenOrCreate));
            BinaryWriter binWriterB = new BinaryWriter(new FileStream(tempB,
FileMode.OpenOrCreate));
            BinaryWriter binWriterC = new BinaryWriter(new FileStream(tempC,
FileMode.OpenOrCreate));
            int cnt = 0;

            while (!StreamEnd(binReader))
            {
                if (cnt % 2 == 0)
                {
                    for (int i = 0; i < partSize; i++)
                    {
                        if (!StreamEnd(binReader))
                            binWriterB.Write(binReader.ReadInt32());
                    }
                }

                else
                {
                    for (int i = 0; i < partSize; i++)
                    {
                        if (!StreamEnd(binReader))
                            binWriterC.Write(binReader.ReadInt32());
                    }
                }
                cnt++;
            }

            binReader.Close();
            binWriterB.Close();
            binWriterC.Close();
        }
        protected void Merge(string filepath, string tempB, string tempC, long
partSize)
        {
            BinaryWriter binWriter = new BinaryWriter(new FileStream(filepath,
FileMode.OpenOrCreate));
            BinaryReader binReaderB = new BinaryReader(new FileStream(tempB,
FileMode.OpenOrCreate));
```

```
            BinaryReader binReaderC = new BinaryReader(new FileStream(tempC,
FileMode.OpenOrCreate));
            int b, c;
            int counterB, counterC;

            while (!StreamEnd(binReaderC))
            {
                counterB = 0;
                counterC = 0;
                b = binReaderB.ReadInt32();
                c = binReaderC.ReadInt32();
                while (counterB < partSize && counterC < partSize)
                {
                    if (b <= c)
                    {
                        binWriter.Write(b);
                        counterB++;
                        if (counterB < partSize)
                            b = binReaderB.ReadInt32();
                        else
                        {
                            while (counterC < partSize)
                            {
                                binWriter.Write(c);
                                counterC++;
                                if (counterC < partSize && !StreamEnd(binReaderC))
                                    c = binReaderC.ReadInt32();
                            }
                        }
                    }
                    else
                    {
                        binWriter.Write(c);
                        counterC++;
                        if (counterC < partSize && !StreamEnd(binReaderC))
                            c = binReaderC.ReadInt32();
                        else
                        {
                            while (counterB < partSize)
                            {
                                binWriter.Write(b);
                                counterB++;
                                if (counterB < partSize)
                                    b = binReaderB.ReadInt32();
                            }
                        }
                    }
                }
                while (!StreamEnd(binReaderB))
                    binWriter.Write(binReaderB.ReadInt32());

                binWriter.Close();
                binReaderB.Close();
                binReaderC.Close();
                File.Delete(tempB);
                File.Delete(tempC);
            }

            protected bool StreamEnd(BinaryReader binReader)
            {
                if (binReader.BaseStream.Position == binReader.BaseStream.Length)
                    return true;
                return false;
            }
```

```
        public void Sort(string filepath, string tempB, string tempC, long
numberOfElements)
        {
            for (long i = 1; i < numberOfElements; i *= 2)
            {
                // i == partSize
                Split(filepath, tempB, tempC, i);
                Merge(filepath, tempB, tempC, i);
            }
        }
    }
}
```

OptimizedSort.cs

```
using Microsoft.VisualBasic.FileIO;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lab1
{
    internal class OptimizedSort : SortAssistant
    {
        private int _bufferSize = 1024 * 1024 * 128 / sizeof(int);

        private void PreSort(string filename, string tempFilename = "temp")
        {
            BinaryReader binReader = new BinaryReader(new FileStream(filename,
FileMode.Open));
            BinaryWriter binWriter = new BinaryWriter(new FileStream(tempFilename,
FileMode.OpenOrCreate));

            int[] intData;
            while (!StreamEnd(binReader))
            {
                intData = FillIntDataArr(binReader, _bufferSize);
                Array.Sort(intData);
                for (int number = 0; number < intData.Length; number++)
                {
                    binWriter.Write(number);
                }
            }
            binReader.Close();
            binWriter.Close();
            File.Delete(filename);
            FileSystem.RenameFile(tempFilename, filename);
            Console.WriteLine("Pre-Sort is done");
        }
        protected void Split(string filepath, string tempB, string tempC, long
partSize)
        {
            BinaryReader binReader = new BinaryReader(new FileStream(filepath,
FileMode.Open));
            BinaryWriter binWriterB = new BinaryWriter(new FileStream(tempB,
FileMode.OpenOrCreate));
            BinaryWriter binWriterC = new BinaryWriter(new FileStream(tempC,
FileMode.OpenOrCreate));
            long counter = 0;

            while ((binReader.BaseStream.Length - binReader.BaseStream.Position) /
sizeof(int) > _bufferSize)
```

```
            {
                if ((int)(counter / partSize % 2) == 0)
                    binWriterB.Write(binReader.ReadBytes(_bufferSize * sizeof(int)));
                else
                    binWriterC.Write(binReader.ReadBytes(_bufferSize * sizeof(int)));
                counter += _bufferSize;
            }

            if ((int)(counter / partSize % 2) == 0)
                binWriterB.Write(binReader.ReadBytes((int)(binReader.BaseStream.Length
- binReader.BaseStream.Position)));
            else
                binWriterC.Write(binReader.ReadBytes((int)(binReader.BaseStream.Length
- binReader.BaseStream.Position)));

            binReader.Close();
            binWriterB.Close();
            binWriterC.Close();
        }

        protected void Merge(string filepath, string tempB, string tempC, long
partSize)
        {
            BinaryWriter binWriter = new BinaryWriter(new FileStream(filepath,
FileMode.OpenOrCreate));
            BinaryReader binReaderB = new BinaryReader(new FileStream(tempB,
FileMode.Open));
            BinaryReader binReaderC = new BinaryReader(new FileStream(tempC,
FileMode.Open));


            while (!StreamEnd(binReaderC))
            {
                long counterB = 0, counterC = 0;
                int bIndex = 0, cIndex = 0;
                int[] b = FillIntDataArr(binReaderB, partSize);
                int[] c = FillIntDataArr(binReaderC, partSize);
                while (true)
                {
                    if (b[bIndex] <= c[cIndex])
                    {
                        WriteNumberToA(b, ref bIndex, ref counterB, binWriter);
                        if (b.Length == bIndex)
                        {
                            if (counterB < partSize)
                            {
                                b = FillIntDataArr(binReaderB, partSize - counterB);
                                bIndex = 0;
                            }
                            else
                            {
                                WriteArrayToA(c, ref cIndex, ref counterC, binWriter);
                                while (counterC < partSize && !StreamEnd(binReaderC))
                                {
                                    c = FillIntDataArr(binReaderC, partSize -
counterC);
                                    cIndex = 0;
                                    WriteArrayToA(c, ref cIndex, ref counterC,
binWriter);
                                }
                                break;
                            }
                        }
                    }
                    else
```

```
                    {
                        WriteNumberToA(c, ref cIndex, ref counterC, binWriter);
                        if (c.Length - cIndex == 0)
                        {
                            if (counterC < partSize && !StreamEnd(binReaderC))
                            {
                                c = FillIntDataArr(binReaderC, partSize - counterC);
                                cIndex = 0;
                            }
                            else
                            {
                                WriteArrayToA(b, ref bIndex, ref counterB, binWriter);
                                while (counterB < partSize)
                                {
                                    b = FillIntDataArr(binReaderB, partSize -
counterB);
                                    bIndex = 0;
                                    WriteArrayToA(b, ref bIndex, ref counterB,
binWriter);
                                }
                                break;
                            }
                        }
                    }
                }
            }
            binWriter.Write(binReaderB.ReadBytes((int)(binReaderC.BaseStream.Length -
binReaderB.BaseStream.Length)));
            binWriter.Close();
            binReaderB.Close();
            binReaderC.Close();
            File.Delete(tempB);
            File.Delete(tempC);
        }
        private int[] FillIntDataArr(BinaryReader binReader, long partSize)
        {
            byte[] binData;
            long count = binReader.BaseStream.Length - binReader.BaseStream.Position;

            if (count <= partSize * sizeof(int) && count <= _bufferSize * sizeof(int))
                binData = binReader.ReadBytes((int)count);
            else if (partSize < _bufferSize)
                binData = binReader.ReadBytes((int)partSize * sizeof(int));
            else
                binData = binReader.ReadBytes((int)_bufferSize * sizeof(int));

            int[] intData = new int[binData.Length / sizeof(int)];
            for (int i = 0; i < intData.Length; i++)
                intData[i] = BitConverter.ToInt32(binData[(i * sizeof(int))..((i + 1)
* sizeof(int))]);

            return intData;
        }

        private void WriteArrayToA(int[] numbers, ref int index, ref long counter,
BinaryWriter binWriter)
        {
            while (numbers.Length > index)
            {
                binWriter.Write(numbers[index]);
                counter++;
                index++;
            }
        }
```

```
        private void WriteNumberToA(int[] numbers, ref int index, ref long counter,
BinaryWriter binWriter)
        {
            binWriter.Write(numbers[index]);
            counter++;
            index++;
        }
        protected bool StreamEnd(BinaryReader binReader)
        {
            if (binReader.BaseStream.Position == binReader.BaseStream.Length)
                return true;
            return false;
        }

        public void Sort(string filepath, string tempB, string tempC, long
numberOfElements)
        {
            PreSort(filepath);
            for (long i = _bufferSize; i < numberOfElements; i *= 2)
            {
                // i == partSize
                Split(filepath, tempB, tempC, i);
                Merge(filepath, tempB, tempC, i);
            }
        }
    }
}
```

InputAssistant.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lab1
{
    internal class InputAssistant
    {
        public int GetFileLength()
        {
            while (true)
            {
                Console.Write("Enter your file length in Mb: ");
                string input = Console.ReadLine();
                if (int.TryParse(input, out int result) && result > 0)
                    return result;
                else
                    Console.Write("Wrong input. Try again. ");
            }
        }

        public string GetFilename()
        {
            while (true)
            {
                Console.Write("Enter your filename: ");
                string result = Console.ReadLine();
                if (!string.IsNullOrEmpty(result))
                    return result;
                else
                    Console.Write("Wrong input. Try again. ");
            }
```

```
        }

        public SortAssistant ChooseClasicOrOptimized()
        {
            while (true)
            {
                Console.Write("Choose Classic (1) or Optimized (0) merge sort
alorithm:  ");
                string input = Console.ReadLine();
                if (input == "1")
                    return new MergeSort();
                else if (input == "0")
                    return new OptimizedSort();
                else
                    Console.Write("Wrong symbol. Try again: ");
            }
        }
    }
}
```

SortAssistant.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lab1
{
    internal interface SortAssistant
    {
        protected void Split(string filepath, string tempB, string tempC, long
partSize, long numberOfElements) { }
        protected void Merge(string filepath, string tempB, string tempC, long
partSize, long numberOfElements) { }
        public void Sort(string filepath, string tempB, string tempC, long
numberOfElements) { }
    }
}
```

ВИСНОВОК

При виконанні даної лабораторної роботи я реалізував один із алгоритмів зовнішнього сортування, а саме алгоритм прямого злиття. Розробив програму за допомогою мови C# та перевірив її при сортуванні багатьох файлів різної розмірності. Також був розроблений оптимізований алгоритм сортування, який допоміг покращити час виконання програми.

Порівнюючи швидкість звичайного та оптимізованого сортування, можна побачити значну перевагу оптимізованого сортування. Для прикладу, файл на 10 МБ був відсортований звичайним алгоритмом за 107311 мс, в той час як оптимізованим – за 752 мс.

Також було проведено тестування оптимізованого алгоритму на сортуванні файлу, розмір якого у 2 рази більший, ніж обсяг ОЗУ комп’ютера. Розмір файлу склав 16гб, його сортування зайняло 2209с, або 36,81 хвилин. Тобто сортування в середньому витрачало 2,3 хвилин на сортування 1 гігабайту даних.

Модифікована версія алгоритму відрізняється від звичайної тим, що для зчитування та запису даних використовує буфер, розмір якого визначено у 128 Мб. Це зменшило кількість викликів до диску, що пришвидшило швидкість роботи алгоритму.

Також до того, як сортувати файл за допомогою сортування прямим злиттям, я роблю сортування чанків у файлі за допомогою вбудованого внутрішнього сортування. Це допомагає впорядкувати дані у файлі й пришвидшити його подальше сортування.