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 № 1з дисципліни
«Проектування алгоритмів»

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

Виконав(ла) *ІП-13 Макарчук Лідія Олександрівна* _____
(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 10Мб, 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 16

Багатофазне сортування

3.1 Псевдокод алгоритму

3.1.1 **Загальний алгоритм**

1. ПОЧАТОК
2. Ініціювати змінні
    2.1. Ввести загальну кількість файлів N
    2.2. Ввести довжину однієї серій у байтах
    2.3. Присвоїти numberInOneRun = довжина серії у байтах / кількість байт у цілому числі
    2.4. Створити N файлів
    2.5. Створити TapesIndexArray розміром N
    2.6. Присвоїти ActualRunsIndexArray = ∅
3. Розділити вхідний файл на серії та відсортувати їх (пункт 3.1.2)
4. Розподілити створені серії між N-1 файлами (пункт 3.1.3)
5. Виконати багатофазне сортування (пункт 3.1.4)
6. Видалити допоміжні файли
7. КІНЕЦЬ

3.1.2 **Створення відсортованих серій**

1. ПОЧАТОК
2. Ініціювати змінні
    2.1. numberBytesInOneRun = 0
    2.2. runNumber = 0
    2.3. totalReadIntNumber = 0
    2.4. fileSizeInBytes = розмір unsortedFile у байтах
    2.5. fileSize = кількість цілих чисел у fileSizeInBytes
    2.6. posInBuff = 0

3. Скопіювати unsortedFile у Tapes[N]
4. Відкрити файл Tapes[N] для читання
5. Зчитати у змінну buff[] максимально можливе число цілих чисел
6. Присвоїти prev = мінімальне можливе значення цілого числа
7. ПОКИ totalReadIntNumber != fileSize:
    7.1. number = buff[posInBuff]
    7.2. posInBuff = posInBuff + 1
    7.3. ЯКЩО number < prev ТО:
        7.3.1. Додати кількість байт у даній серії (numberBytesInOneRun) у массив довжин серій Tapes[N].lenghtOfRuns
        7.3.2. runNumber = runNumber + 1
        7.3.3. numberBytesInOneRun = кількість байт у цілому числі
    7.4. ІНАКШЕ
        7.4.1. numberBytesInOneRun = numberBytesInOneRun + кількість байт у цілому числі
    7.5. prev = number
    7.6. totalReadIntNumber = totalReadIntNumber + 1
    7.7. ЯКЩО posInBuff == buff.Lenght ТО
        7.7.1. ЯКЩО buff.Lenght < fileSize ТО
            7.7.1.1. Зчитати у змінну buff[] максимально можливе число цілих чисел
        7.7.2. posInBuff = 0
8. ЯКЩО numberBytesInOneRun > 0 ТО
    8.1. Додати кількість байт у даній серії (numberBytesInOneRun) у массив довжин серій Tapes[N].lenghtOfRuns
    8.2. runNumber = runNumber + 1
9. Закрити файл Tapes[N] для читання
10. КІНЕЦЬ

3.1.3 **Початкове розподілення серій між файлами**

1. ПОЧАТОК
2. Логічно розподілити серії між файлами
   - 2.1. Присвоїти fibonacciNumbers[] числа Фібаначчі порядку N-2, які у сумі становлять >= кількості серій (runNumber)
   - 2.2. Присвоїти level = кількість ітерацій при відніманні від fibonacciNumbers[] найменшого з них поки всі fibonacciNumbers[] крім одного не будуть <= 0
   - 2.3. Присвоїти perfectRunNumber = сума fibonacciNumbers[]
   - 2.4. Присвоїти emptyRunNumbers = perfectRunNumber – runNumber
   - 2.5. Цикл проходу по файлах (i = 1, 2, …, N-1)
     - 2.5.1. Tapes[i].runNumber = fibonacciNumbers[i]
     - 2.5.2. Tapes[i].totalNumber = fibonacciNumbers[i]
   - 2.6. Присвоїти i = emptyRunNumbers
   - 2.7. ПОКИ i > 0
     - 2.7.1. Присвоїти j = 0
     - 2.7.2. ПОКИ i > 0 І j < N-1
       - 2.7.2.1. ЯКЩО Tapes[j].runNumber > 0, ТО:
         - 2.7.2.1.1. Tapes[j].runNumber = Tapes[j].runNumber - 1
         - 2.7.2.1.2. Tapes[j].dummyRunNumber = Tapes[j].dummyRunNumber + 1
         - 2.7.2.1.3. i = i – 1
       - 2.7.2.2. j = j + 1
3. Фізично розподілити серії між фійлами
   - 3.1. Присвоїти currRun = 1
   - 3.2. Відкрити файли
     - 3.2.1. Почати читати файл Tapes[N]
     - 3.2.2. Почати запис до файлів Tapes[i] (i=1, 2, …, N-1)
   - 3.3. Цикл проходу по файлах для запису (i=1, 2, …, N-1)

3.3.1. Цикл проходу по всіх Tapes[i].runNumber (j = 1, 2, …, Tapes[i].runNumber)

3.3.1.1. Присвоїти numberOfNumbers = Tapes[N].lengthOfRuns[currRuns] / кількість байтів у цілому числі

3.3.1.2. Присвоїти currRuns = currRuns + 1

3.3.1.3. Зчитати з Tapes[N] до масиву buff[] numberOfNumbers цілих чисел

3.3.1.4. Записати масив buff[] до файлу Tapes[i]

3.3.1.5. Присвоїти writtenBytes = кількість записаних байт з buff[]

3.3.1.6. Додати writtenBytes до Tapes[i].lenghtOfRuns

3.3.2. Закрити файл Tapes[i] для запису

3.3.3. Відкрити файл Tapes[i] для читання

3.4. Закрити файл Tapes[N] для читання

4. КІНЕЦЬ

3.1.4 **Алгоритм багатофазного сортування**

1. ПОЧАТОК
2. ПОКИ level != 0:
   2.1. Присвоїти finalTapeIndex = TapesIndexArray[N]
   2.2. currRuns = найменше totalNumber серед файлів Tapes[TapesIndexArray [i]] (i = 1, 2, …, N-1)
   2.3. TapeIndex = номер файла у Tapes[TapesIndexArray [i]], що містить найменше totalNumber серед файлів Tapes[TapesIndexArray [i]] (i = 1, 2, …, N-1)
   2.4. Змінити дані Tapes[TapesIndexArray [N]] на початкові дані:
      2.4.1. Tapes[TapesIndexArray [N]].dummyRunNumber = 0
      2.4.2. Tapes[TapesIndexArray [N]].runNumber = 0
      2.4.3. Tapes[TapesIndexArray [N]].totalRunNumber = 0

2.4.4. Tapes[TapesIndexArray [N]].lengthOfRuns = ∅

2.4.5. Закрити файл Tapes[TapesIndexArray [N]] для читання

2.4.6. Видалити вміст файлу Tapes[TapesIndexArray [N]]

2.4.7. Відкрити файл Tapes[TapesIndexArray[N]] для запису

2.5. ПОКИ currRuns != 0

2.5.1. Цикл проходу по файлах Tapes[TapesIndexArray[i]] (i = 1, 2, …, N-1)

2.5.1.1. ЯКЩО Tapes[TapesIndexArray [i]].dummyRunNumber > 0:

2.5.1.1.1. Зменшити Tapes[TapesIndexArray [i]].dummyRunNumber на 1

2.5.1.1.2. Зменшити Tapes[TapesIndexArray[i]]. totalRunNumber на 1

2.5.1.2. ІНАКШЕ ЯКЩО Tapes[TapesIndexArray[i]].runNumber > 0:

2.5.1.2.1. Додати індекс TapesIndexArray[i] до масиву індексів ActualRunsIndexArray[]

2.5.2. ЯКЩО довжина масиву ActualRunsIndexArray[] == 0:

2.5.2.1. Збільшити Tapes[TapesIndexArray[i]].dummyRunNumber на 1

2.5.2.2. Збільшити Tapes[TapesIndexArray[N]].totalRunNumber на 1

2.5.2.3. Зменшити currRuns на 1

2.5.3. ІНАКШЕ

2.5.3.1. Виконати злиття реальних серій (пункт 3.1.5)

2.5.3.2. Зменшити currRuns на 1

2.6. Закрити файл Tapes[TapesIndexArray[N]] для запису

2.7. Відкрити файл Tapes[TapesIndexArray[N]] для читання

2.8. Поміняти місцями TapesIndexArray[tapeIndex] і TapesIndexArray[N]

2.9. Зменшити level на 1

3. Цикл проходу по файлах (i = 1, 2, …, N-1):
   3.1. Закрити файл TapesIndexArray[i] для читання
4. Закрити файл TapesIndexArray[N] для запису
5. КІНЕЦЬ

3.1.5 **Злиття реальних серій**

1. Ініціювати змінні:
   1.1. Цикл проходу по файлах у масиві ActualRunsIndexArray[] (i = 1, 2, …, довжина ActualRunsIndexArray[]):
      1.1.1. Зчитати число з файла Tapes[TapesIndexArray[i]] у numbers[i]
      1.1.2. Присвоїти writtenInts[i] = 1
   1.2. Присвоїти totalWrittenNumberOfInts = 0
   1.3. ПОКИ довжина ActualRunsIndexArray[] != 0
      1.3.1. Цикл проходу по numbers[i], де i = 1, 2, …, довжина numbers, min = найменше значення серед numbers[i], minTapeIndex = індекс і найменшого значення min
      1.3.2. Записати знайдене значення min до файлу Tapes[ActualRunsIndexArray[N]]
      1.3.3. Збільшити totalWrittenNumberOfInts на 1
      1.3.4. ЯКЩО writtenInts[minTapeIndex] == довжині поточної серії Tapes[ActualRunsIndexArray[minTapeIndex]], поділеної на розмір цілого числа у байтих ТО:
         1.3.4.1. Зменшити Tapes[ActualRunsIndexArray[minTapeIndex]].runNumber на 1
         1.3.4.2. Зменшити Tapes[ActualRunsIndexArray[minTapeIndex]].totalNumber на 1

1.3.4.3. Видалити довжину поточної серії з Tapes[ActualRunsIndexArray[minTapeIndex]].lengthOfRuns

1.3.4.4. Присвоїти ActualRunsIndexArray[minTapeIndex] = ActualRunsIndexArray[останній елемент]

1.3.4.5. Видалити ActualRunsIndexArray[останній елемент]

1.3.4.6. Присвоїти numbers[minTapeIndex] = numbers[останній елемент]

1.3.4.7. Видалити numbers[останній елемент]

1.3.4.8. Присвоїти writtenInts[minTapeIndex] = writtenInts[останній елемент]

1.3.4.9. Видалити writtenInts[останній елемент]

1.3.5. ІНАКШЕ

1.3.5.1. Зчитати число з файла Tapes[TapesIndexArray [minTapeIndex]] у numbers[minTapeIndex]

1.3.5.2. Збільшити writtenInts[minTapeIndex] на 1

1.4. Додати до Tapes[TapesIndexArray[minTapeIndex]].lengthOfRuns totalWrittenNumberOfInts * кількість байтів в 1 цілому числі

1.5. Збільшити Tapes[TapesIndexArray[minTapeIndex]].runNumber на 1

1.6. Збільшити Tapes[TapesIndexArray[minTapeIndex]].totalRunNumber на 1

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

//Модуль PolyphaseMerge

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Linq;
using System.Numerics;
```

```
using System.Reflection.Metadata.Ecma335;
using System.Runtime.Intrinsics;
using System.Text;
using System.Threading.Tasks;

namespace laba1_1
{
    /* class for distribution all the runs between Tapes and merging them into one */
    internal class PolyphaseMerge
    {
        private int N ;    //number of Tapes
        private List<Tape> Tapes; //files
        private List<int> TapesIndexArray;    //stores indices for swapping input/output Tapes
        private List<int> ActualRunsIndexArray;   //stores indices of tapes with real runs that will be merged
        private int level;
        private int finalTapeIndex;
        private long maxBytesInOneRun;


        public PolyphaseMerge(int TapesNumber, long maxBytesInOneRun1)   /*      Constructor */
        {
            N = TapesNumber;
            Tapes = new List<Tape>(new Tape[N]);
            for(int i=0;i<N;i++)
                Tapes[i] = new Tape();
            TapesIndexArray = new List<int>(new int[N]);
            ActualRunsIndexArray = new List<int>();
            for (int i = 0; i < N; i++)
            { TapesIndexArray[i] = i; }
            level = 0;
            finalTapeIndex = 0;
            maxBytesInOneRun = maxBytesInOneRun1;
        }

        public void PolyphaseMergeSort(string unsortedFile, ref string sortedFile, int mode)
        {
            int runNumber;
            if (mode == 1)
                runNumber = createRuns(unsortedFile, maxBytesInOneRun);
            else
                runNumber = createRunsOptimized(unsortedFile, maxBytesInOneRun);
            DistributeRunNumber(runNumber);
            InitialDistribution();
            Polyphase(mode);

            deleteTempFiles();
            renameFinalFile(ref sortedFile);
        }

        public int calculateRunNumber(string filePath, ref long numberInOneRun, long bytesInOneRun)   //calculates total run
number and how many numbers will be in each run
        {
            long fileSize = new System.IO.FileInfo(filePath).Length;
            int runNumber = (int)(fileSize / bytesInOneRun);
            numberInOneRun = bytesInOneRun / sizeof(int);
            maxBytesInOneRun = numberInOneRun * sizeof(int);
            return runNumber;
        }
        public int createRuns(string filePath, long bytesInOneRun) //divides initial file into runs and sorts every of them
        {
            long numberBytesInOneRun = 0;
            int runNumber = 0;
            long totalReadIntNumber = 0;
            long fileSizeInBytes = new System.IO.FileInfo(filePath).Length;
            long fileSize = fileSizeInBytes / sizeof(int);
            const int bytesInOneInt = sizeof(int);
```

```
            Tape sortedRunsFile = new Tape(filePath);
            sortedRunsFile.StartRead();
            Tapes[N - 1].StartWrite();
            FileManager.copyFile(sortedRunsFile.binaryReader, Tapes[N - 1].binaryWriter, fileSizeInBytes, bytesInOneRun);
//copies file content to Tape[N-1] file
            sortedRunsFile.binaryReader.Close();
            Tapes[N - 1].binaryWriter.Close();

            Tapes[N - 1].StartRead();
            int[] buff = FileManager.readArrayOfInts(Tapes[N-1].binaryReader, bytesInOneRun/ bytesInOneInt);
            int posInBuff = 0;
            int prev = Int32.MinValue;
            while(totalReadIntNumber != fileSize)
            {
               int number = buff[posInBuff++];
               if (number < prev)
               {
                  Tapes[N - 1].AddLengthOfRuns(numberBytesInOneRun);
                  runNumber++;
                  numberBytesInOneRun = bytesInOneInt;
               }
               else
               {
                  numberBytesInOneRun+= bytesInOneInt;
               }
               prev = number;
               totalReadIntNumber++;
               if(posInBuff == buff.Length)
               {
                  if(totalReadIntNumber < fileSize)//
                     buff = FileManager.readArrayOfInts(Tapes[N-1].binaryReader, bytesInOneRun / bytesInOneInt);
                  posInBuff = 0;
               }
            }
            if (numberBytesInOneRun>0)
            {
               Tapes[N - 1].AddLengthOfRuns(numberBytesInOneRun);
               runNumber++;
            }
            Tapes[N - 1].binaryReader.Close();

            return runNumber;
         }
public int createRunsOptimized(string filePath, long bytesInOneRun) //divides initial file into runs and sorts every of them
         {
            long numberInOneRun = 0;
            int runNumber = calculateRunNumber(filePath, ref numberInOneRun, bytesInOneRun);

            Tape sortedRunsFile = new Tape(filePath);
            sortedRunsFile.StartRead();
            Tapes[N - 1].StartWrite();

            long usedBytes = 0;

            for (int i = 0; i < runNumber - 1; i++)
            {
               int[] buff = FileManager.readArrayOfInts(sortedRunsFile.binaryReader, numberInOneRun);   //reading the run as
bytes
               Array.Sort(buff);
               long writtenBytes = FileManager.writeArrayOfInts(Tapes[N - 1].binaryWriter, ref buff);
               Tapes[N - 1].AddLengthOfRuns(writtenBytes);
               usedBytes += writtenBytes;
            }
```

```
            long fileSize = new System.IO.FileInfo(filePath).Length;
            long newBuffSize = fileSize - usedBytes;
            long newNumberInOneRun = newBuffSize / sizeof(int);

            int[] lastBuff = FileManager.readArrayOfInts(sortedRunsFile.binaryReader, newNumberInOneRun);
            Array.Sort(lastBuff);
            long newWrittenBytes = FileManager.writeArrayOfInts(Tapes[N - 1].binaryWriter, ref lastBuff);
            Tapes[N - 1].AddLengthOfRuns(newWrittenBytes);

            sortedRunsFile.binaryReader.Close();
            Tapes[N - 1].binaryWriter.Close();

            return runNumber;
        }
        public void DistributeRunNumber(int runNumber)//distributes the runs between the tapes' runNumbers and
dummyRunNumbers, is based on fibonacci numbers
        {
            List<int> fibonacciNumbers = FibonacciNumbers.calculate_runs_distribution(N, runNumber, ref level);
            int perfectRunNumber = 0;
            int i;
            for (i = 0; i < fibonacciNumbers.Count(); i++)
                perfectRunNumber += fibonacciNumbers[i];
            int emptyRunNumbers = perfectRunNumber - runNumber;
            for (i = 0; i < N - 1; i++)
            {
                Tapes[i].runNumber = fibonacciNumbers[i];   //setting runs for tapes
                Tapes[i].totalRunNumber = fibonacciNumbers[i];
            }
            i = emptyRunNumbers;
            while (i > 0)   //distribute dummy runs evenly between the tapes
            {
                int j = 0;
                while (i > 0 && j < N - 1)
                {
                    if (Tapes[j].runNumber > 0)
                    {
                        Tapes[j].runNumber--;
                        Tapes[j].dummyRunNumber++;
                        i--;
                    }
                    j++;
                }
            }
        }

        public void InitialDistribution() //reads sorted runs from file and distributes it between Tapes
        {
            Tapes[N - 1].StartRead();
            int currRun = 0;
            int i;
            for (i = 0; i < N - 1; i++)
                Tapes[i].StartWrite();

            for (i = 0; i < N - 1; i++)
            {
                for (int j = 0; j < Tapes[i].runNumber; j++)
                {
                    long numberOfNumbers = Tapes[N-1].lengthOfRuns[currRun] / sizeof(int);
                    currRun++;
                    int[] buff = FileManager.readArrayOfInts(Tapes[N - 1].binaryReader, numberOfNumbers);
                    long writtenBytes = FileManager.writeArrayOfInts(Tapes[i].binaryWriter, ref buff);
                    Tapes[i].AddLengthOfRuns(writtenBytes);
                }
                Tapes[i].binaryWriter.Close();
                Tapes[i].StartRead();
            }
```

```
            Tapes[N - 1].binaryReader.Close();
        }

        public void Polyphase(int mode)   //for merging the sorted runs
        {
            while (level != 0)
            {
                finalTapeIndex = TapesIndexArray[N - 1];
                int tapeIndex = -1;
                int currRuns = pickSmallestTotalNumber(ref tapeIndex);  //total number of runs in the last tape (also the smallest
number of runs)
                Tapes[TapesIndexArray[N - 1]].Reset();
                Tapes[TapesIndexArray[N - 1]].binaryReader.Close();
                Tapes[TapesIndexArray[N - 1]].StartWriteTrunc();
                while (currRuns != 0)
                {
                    for (int i = 0; i < N - 1; i++)
                    {
                        if (Tapes[TapesIndexArray[i]].dummyRunNumber > 0)
                        {
                            Tapes[TapesIndexArray[i]].dummyRunNumber--; //use dummy run for merging
                            Tapes[TapesIndexArray[i]].totalRunNumber--;
                        }
                        else if (Tapes[TapesIndexArray[i]].runNumber > 0)
                        {
                            ActualRunsIndexArray.Add(TapesIndexArray[i]);
                        }
                    }
                    if (ActualRunsIndexArray.Count() == 0)
                    {
                        Tapes[TapesIndexArray[N - 1]].dummyRunNumber++; //merge dummy runs
                        Tapes[TapesIndexArray[N - 1]].totalRunNumber++;
                        currRuns--;
                    }
                    else
                    {
                        if (mode == 1)
                            mergeRuns();    //mergeing
                        else
                            mergeRunsOptimized(maxBytesInOneRun);
                        currRuns--;
                    }
                }
                Tapes[TapesIndexArray[N - 1]].binaryWriter.Close();
                Tapes[TapesIndexArray[N - 1]].StartRead();  //start reading recently merged Tape
                int temp = TapesIndexArray[tapeIndex];  //swapping the merged Tape and the used Tape
                TapesIndexArray[tapeIndex] = TapesIndexArray[N - 1];
                TapesIndexArray[N - 1] = temp;
                level--;
            }
            for (int i = 0; i < N-1; i++)
                Tapes[i].binaryReader.Close();
            Tapes[N-1].binaryWriter.Close();
        }
    public void mergeRuns() //merging the real runs
        {
            List<int> numbers = new List<int>(new int[ActualRunsIndexArray.Count()]);
            List<int> writtenInts = new List<int>(new int[ActualRunsIndexArray.Count()]);
            for (int i = 0; i < ActualRunsIndexArray.Count(); i++)
            {
                numbers[i] = Tapes[ActualRunsIndexArray[i]].binaryReader.ReadInt32(); //reading a number from each tape
                writtenInts[i] = 1;
            }
            long totalWrittenNumberOfInts = 0;
            while (ActualRunsIndexArray.Count() != 0)
```

```
        {
            int min = numbers[0];
            int minTapeIndex = 0;
            for (int i = 1; i < numbers.Count(); i++)
            {
                if (numbers[i] < min)   //finding the smallest number
                {
                    min = numbers[i];
                    minTapeIndex = i;
                }
            }
            Tapes[TapesIndexArray[N - 1]].binaryWriter.Write(BitConverter.GetBytes(min));//writing this number to a source
tape
            totalWrittenNumberOfInts++;
            if (writtenInts[minTapeIndex] == Tapes[ActualRunsIndexArray[minTapeIndex]].lengthOfRuns[0]/sizeof(int))
//checking if the run of the tape with min number hasn't been used completely
            {
                /*      deleting the tape if its run has been used*/
                Tapes[ActualRunsIndexArray[minTapeIndex]].runNumber--;
                Tapes[ActualRunsIndexArray[minTapeIndex]].totalRunNumber--;
                Tapes[ActualRunsIndexArray[minTapeIndex]].remove_run_position();
                ActualRunsIndexArray[minTapeIndex] = ActualRunsIndexArray[ActualRunsIndexArray.Count() - 1];
                numbers[minTapeIndex] = numbers[numbers.Count() - 1];
                ActualRunsIndexArray.RemoveAt(ActualRunsIndexArray.Count - 1);
                numbers.RemoveAt(numbers.Count - 1);
                writtenInts[minTapeIndex] = writtenInts[writtenInts.Count - 1];
                writtenInts.RemoveAt(writtenInts.Count - 1);
            }
            else
            {
                numbers[minTapeIndex] = Tapes[ActualRunsIndexArray[minTapeIndex]].binaryReader.ReadInt32();  //writing a
new number
                writtenInts[minTapeIndex]++;
            }
        }
        Tapes[TapesIndexArray[N - 1]].AddLengthOfRuns(totalWrittenNumberOfInts*sizeof(int));
        Tapes[TapesIndexArray[N - 1]].runNumber++;
        Tapes[TapesIndexArray[N - 1]].totalRunNumber++;
    }

    public int pickSmallestTotalNumber(ref int tapeIndex)
    {
        int smallest = Int32.MaxValue;
        for (int i = 0; i < N; i++)
        {
            if (Tapes[TapesIndexArray[i]].totalRunNumber > 0 && Tapes[TapesIndexArray[i]].totalRunNumber < smallest)
            {
                smallest = Tapes[TapesIndexArray[i]].totalRunNumber;
                tapeIndex = i;
            }
        }
        return smallest;
    }
    public void deleteTempFiles() //delets all temp files
    {
        for (int i = 0; i < N; i++)
        {
            Tapes[i].closeReaders();
            if (i != finalTapeIndex)
                Tapes[i].destroy();
        }
    }
    public void renameFinalFile(ref string newFileName)
    {
        Tapes[finalTapeIndex].closeReaders();
        bool isRenamed = false;
```

```
            do
            {
                isRenamed = FileManager.renameFile(Tapes[finalTapeIndex].fileName, ref newFileName);
            }
            while (!isRenamed);
        }

        public void mergeRunsOptimized(long maxNumberOfBytes)
        {
            List<int[]> buffers = new List<int[]>();
            List<int> indexInBuffs = new List<int>(new int[ActualRunsIndexArray.Count()]);
            long writtenIntsToSourceFile = 0;
            int[] sourceBuff = new int[maxNumberOfBytes/sizeof(int)];
            for (int i = 0; i < ActualRunsIndexArray.Count(); i++)
            {
                long currBuffSize = Math.Min(maxNumberOfBytes, Tapes[ActualRunsIndexArray[i]].lengthOfRuns[0]);
                Tapes[ActualRunsIndexArray[i]].lengthOfRuns[0] -= currBuffSize;
                buffers.Add(FileManager.readArrayOfInts(Tapes[ActualRunsIndexArray[i]].binaryReader,
currBuffSize/sizeof(int)));
                indexInBuffs[i] = 0;
            }
            long totalWrittenNumberOfInts = 0;
            while (ActualRunsIndexArray.Count() != 0)
            {
                int min = buffers[0][indexInBuffs[0]];
                int minTapeIndex = 0;
                for (int i = 1; i < buffers.Count(); i++)
                {
                    if (buffers[i][indexInBuffs[i]] < min)   //finding the smallest number
                    {
                        min = buffers[i][indexInBuffs[i]];
                        minTapeIndex = i;
                    }
                }
                sourceBuff[writtenIntsToSourceFile] = min;  //writing to source buff
                writtenIntsToSourceFile++;
                if(writtenIntsToSourceFile == sourceBuff.Length)    //flush the source buff
                {
                    FileManager.writeArrayOfInts(Tapes[TapesIndexArray[N - 1]].binaryWriter, ref sourceBuff);
                    writtenIntsToSourceFile = 0;
                }
                totalWrittenNumberOfInts++;
                indexInBuffs[minTapeIndex]++;
                if (indexInBuffs[minTapeIndex] == buffers[minTapeIndex].Length)//check if buff is not empty
                {
                    if(Tapes[ActualRunsIndexArray[minTapeIndex]].lengthOfRuns[0] == 0)  //if run has been used
                    {
                        Tapes[ActualRunsIndexArray[minTapeIndex]].runNumber--;
                        Tapes[ActualRunsIndexArray[minTapeIndex]].totalRunNumber--;
                        Tapes[ActualRunsIndexArray[minTapeIndex]].remove_run_position();
                        ActualRunsIndexArray[minTapeIndex] = ActualRunsIndexArray[ActualRunsIndexArray.Count() - 1];
                        ActualRunsIndexArray.RemoveAt(ActualRunsIndexArray.Count - 1);

                        buffers[minTapeIndex] = (int[])buffers[buffers.Count() - 1].Clone();
                        buffers.RemoveAt(buffers.Count - 1);

                        indexInBuffs[minTapeIndex] = indexInBuffs[indexInBuffs.Count - 1];
                        indexInBuffs.RemoveAt(indexInBuffs.Count - 1);
                    }
                    else
                    {
                        long currBuffSize = Math.Min(maxNumberOfBytes,
Tapes[ActualRunsIndexArray[minTapeIndex]].lengthOfRuns[0]);
                        Tapes[ActualRunsIndexArray[minTapeIndex]].lengthOfRuns[0] -= currBuffSize;
                        buffers[minTapeIndex] =
FileManager.readArrayOfInts(Tapes[ActualRunsIndexArray[minTapeIndex]].binaryReader, currBuffSize / sizeof(int));
```

```
                    indexInBuffs[minTapeIndex] = 0;
                }
            }
        }
        int[] lastInts = new int[writtenIntsToSourceFile];
        Array.Copy(sourceBuff, 0, lastInts, 0, writtenIntsToSourceFile);
        FileManager.writeArrayOfInts(Tapes[TapesIndexArray[N - 1]].binaryWriter, ref lastInts);

        Tapes[TapesIndexArray[N - 1]].AddLengthOfRuns(totalWrittenNumberOfInts * sizeof(int));
        Tapes[TapesIndexArray[N - 1]].runNumber++;
        Tapes[TapesIndexArray[N - 1]].totalRunNumber++;
    }
  }
}
```

//Модуль Tape

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

namespace laba1_1
{
  internal class Tape
  {
    public string fileName;
    public int runNumber;
    public int dummyRunNumber;
    public int totalRunNumber;
    public FileStream fileObject;
    public List<long> lengthOfRuns;  //in bytes
    public BinaryReader binaryReader;
    public BinaryWriter binaryWriter;

    /* Constructors/Destructors */
    public Tape()
    {
      TapeNumber++;
      fileName = "Tape" + TapeNumber.ToString() + ".txt";
      runNumber = 0;
      dummyRunNumber = 0;
      totalRunNumber = 0;
      lengthOfRuns = new List<long>();
    }
    public Tape(string filePath)
    {
      fileName = filePath;
      runNumber = 0;
      dummyRunNumber = 0;
      totalRunNumber = 0;
      lengthOfRuns = new List<long>();
    }

    /* functions */
    public static int TapeNumber;
    public void destroy() //for deleting the fileObject
    {
      File.Delete(fileName);
    }

    public void closeReaders()
    {
      binaryReader.Close();
```

```
            binaryReader.Close();
        }
        public void Reset()  //set the current position at the start of the file
        {
            dummyRunNumber = 0;
            runNumber = 0;
            totalRunNumber = 0;
            lengthOfRuns.Clear();
        }
        public void StartRead()
        {
            fileObject = new FileStream(fileName, FileMode.Open);
            binaryReader = new BinaryReader(fileObject);
        }
        public void StartWrite()
        {
            fileObject = new FileStream(fileName, FileMode.OpenOrCreate);
            binaryWriter = new BinaryWriter(fileObject);
        }

        public void StartWriteTrunc()
        {
            fileObject = new FileStream(fileName, FileMode.Truncate);
            binaryWriter = new BinaryWriter(fileObject);
        }
        public void AddLengthOfRuns(long runLenght) //adds to end of runs last writed position
        {
            lengthOfRuns.Add(runLenght);
        }
        public void remove_run_position()
        {
            lengthOfRuns.RemoveAt(0);
        }
    }
}
```

//Модуль FibonacciNumbers

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;
using System.Reflection.Emit;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.Intrinsics;

namespace laba1_1
{
    internal class FibonacciNumbers
    {
        public static List<int> calculate_runs_distribution(int tapeNumbers, int runNumber, ref int level)
        {
            int exponent = tapeNumbers - 2;
            List<int> fibonacciNumbers = new List<int>(new int[exponent + 1]);
            fibonacciNumbers[exponent] = 1;
            int i = exponent;

            while (fibonacciNumbers[i] < runNumber)
            {
                int nextNumber = 0;
                for (int j = 0; j <= exponent; j++)
                {
                    nextNumber += fibonacciNumbers[i - j];
                }
```

```
            fibonacciNumbers.Add(nextNumber);
            i++;
        }

        List<int> distribution_numbers = fibonacciNumbers.GetRange(i - 1 - exponent, tapeNumbers - 1);

        level = calculate_level(distribution_numbers);
        return distribution_numbers;

    }

    public static int calculate_level(List<int> fibonacciNumbers)
    {
        List<int> temp = new List<int>(fibonacciNumbers);

        int k = 0;
        while (temp.Count() > 1)
        {
            int index = 0;
            int smallest = find_smallest(temp, ref index);
            if (smallest < 0)
                break;
            for (int i = 0; i < temp.Count(); i++)
                temp[i] -= smallest;
            temp[index] = smallest;

            List<int> newTemp = new List<int>();
            for (int i = 0; i < temp.Count(); i++)
            {
                if (temp[i] > 0)
                    newTemp.Add(temp[i]);
            }
            temp.Clear();
            temp = new List<int>(newTemp);
            k++;
        }
        return k;

    }
    public static int find_smallest(List<int> fibonacciNumbers, ref int index)
    {
        int smallest = Int32.MaxValue;
        for (int i = 0; i < fibonacciNumbers.Count(); i++)
        {
            if (fibonacciNumbers[i] > 0 && fibonacciNumbers[i] < smallest)
            {
                smallest = fibonacciNumbers[i];
                index = i;
            }
        }
        return smallest;
    }
  }
}
```

ВИСНОВОК

При виконанні даної лабораторної роботи я реалізувала один із алгоритмів зовнішнього сортування, а саме багатофазне сортування. Я розробила програмну реалізацію даного алгоритму за допомогою мови C# та відсортувала файл розміром 12228 байт, що становить майже 12 мегабайт. На сортування було витрачено 11 хвилин 20.936 секунди, при цьому використовувалося 3 файли. Також я створила модифіковану версію алгоритму. За її допомогою цей же файл було відсортовано за 0.544 при таких параметрах: кількість файлів = 3, розмір однієї серії = 3 мегабати.

Також за допомогою використання модифікованої версії програми був відсортований файл, розмір якого у 2 рази більший, ніж обсяг оперативної пам’яті ПК, тобто 32ГБ. Сортування тривало 1 годину 26 хвилин при таких параметрах: кількість файлів – 3, кількість мегабайт у одній серії – 400. Таким чином я досягла швидкості сортування 2,6875 хвилин на 1 гігабайт.

Модифікована версія програми відрізняється від звичайної записом та читанням з файлів, а також збільшенням розміру серій та їх попереднє сортування.

По-перше, звичайна версія зчитувала по одному числу з файла, а потім записувала це число до іншого. Модифікація полягала у тому, аби зчитати массив 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 записувала ці числа у масив цілих чисел, перетворювала даний масив у масив байтів і вже потім записувала всі байти одразу до файлу.

По-друге, замість того, аби вважати серією впорядковану підмножину чисел у вхідному файлі, я даю можливість користувачу самому визначити довжину серії. Таким чином я розділяю файл на серії заданої довжини, сортую їх та знову записую у файл. Для файлів з великим розміром дану довжину варто визначати щонайменше 100 мегабайт.

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. Після09.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- програмна реалізація алгоритму – 40%;
- програмна реалізація модифікацій – 40%;
- висновок – 5%.