**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-12 Єльчанінов Артем Юрійович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc109342184)

[2 ЗаВдання 4](#_Toc109342185)

[3 Виконання 6](#_Toc109342186)

[3.1 Псевдокод алгоритму 6](#_Toc109342187)

[3.2 Програмна реалізація алгоритму 6](#_Toc109342188)

[3.2.1 Вихідний код 6](#_Toc109342189)

[Висновок 7](#_Toc109342190)

[Критерії оцінювання 8](#_Toc109342191)

# Мета лабораторної роботи

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

# 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 |

# Виконання

## Псевдокод алгоритму

**Sort(fileName, sortedFileName, helpFilesCount = 3):**

**Begin**

**if** helpFilesCount < 2**:**

**do**

throw new ArgumentException(null, nameof(helpFilesCount))

**end if**

BHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"B{i}.txt").ToArray()

CHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"C{i}.txt").ToArray()

SplitFile(fileName, BHelpFiles)

SortHelper(BHelpFiles, CHelpFiles, out sortedFileName)

**End**

**SortHelper(BHelpFiles, CHelpFiles, fileName)**:

**Begin**

readers = BHelpFiles.Select(f => new StreamReader(f)).ToList()

readers.Where(r => r.EndOfStream).ToList().ForEach(r =>

{

r.Dispose();

readers.Remove(r);

})

**if** (readers.Count == 1) **do:**

fileName = ((FileStream)readers.First().BaseStream).Name

**return**

**end if**

writers = CHelpFiles.Select(f => new StreamWriter(f, append: false)).ToList()

currentWriter = writers.First()

currentReader = readers.First()

nums = new List<int>()

nextNums = new List<int>()

readerAndPrevNum = readers.ToDictionary(r => r, \_ => int.MinValue)

**while** (readers.Count != 0) **do:**

**while** (currentReader.EndOfStream) **do:**

readerToRemove = currentReader

currentReader = readers.NextAfter(currentReader)

readers.Remove(readerToRemove);

readerAndPrevNum.Remove(readerToRemove

readerToRemove.Dispose()

**if** (readers.Count == 0) **do:**

break

**end if**

**end while**

**if** (readers.Count == 0) **do:**

nums.Sort()

**foreach** (n in nums) **do:**

currentWriter.WriteLine(n)

**end foreach**

currentWriter = writers.NextAfter(currentWriter)

nextNums.Sort()

**foreach** (int n in nextNums) **do:**

currentWriter.WriteLine(n)

**end foreach**

break

**end if**

num = int.Parse(currentReader.ReadLine()!)

**if** (num >= readerAndPrevNum[currentReader]) **do:**

nums.Add(num)

readerAndPrevNum[currentReader] = num

**else:**

nextNums.Add(num)

readerAndPrevNum[currentReader] = num

currentReader = readers.NextAfter(currentReader)

**end if**

**if** (nextNums.Count >= readers.Count) **do:**

nums.Sort()

**foreach** (int n in nums) **do:**

currentWriter.WriteLine(n)

**end foreach**

currentWriter = writers.NextAfter(currentWriter)

nums.Clear()

nums.AddRange(nextNums)

nextNums.Clear()

**end if**

**end while**

readers.ForEach(r => r.Dispose())

writers.ForEach(w => w.Dispose())

SortHelper(CHelpFiles, BHelpFiles, out fileName)

**End**

## Програмна реалізація алгоритму

### Вихідний код

**Program.cs:**

using System.Diagnostics;

namespace lab1 {

internal class Program {

private static string fileName = "";

private static IFileGenerator fileGenerator = null!;

private static IFileSorter fileSorter = null!;

private const int linesCount = 100000000;

public static void Main(string[] args) {

Console.Write(new string('-', Console.BufferWidth));

Console.WriteLine("Normal");

ClassicalAlgorithm();

Console.Write(new string('-', Console.BufferWidth));

Console.WriteLine("Modified");

ModifiedAlgorithm();

}

private static void ClassicalAlgorithm() {

fileName = @"TextFile.txt";

fileGenerator = new TextFileGenerator();

fileGenerator.GenerateBySize(fileName, 10);

Console.WriteLine("Generated");

fileSorter = new TextFileSorter();

Stopwatch stopWatch = Stopwatch.StartNew();

fileSorter.Sort(fileName, out string sortedFileName);

stopWatch.Stop();

Console.WriteLine($"Sorted, file: {sortedFileName}, seconds: {stopWatch.Elapsed.TotalSeconds}");

}

private static void ModifiedAlgorithm() {

fileName = @"BinaryFile.dat";

fileGenerator = new BinaryFileGenerator();

fileGenerator.GenerateByLinesCount(fileName, linesCount);

Console.WriteLine("Generated");

fileSorter = new BinaryFileSorter();

Stopwatch stopWatch = Stopwatch.StartNew();

((BinaryFileSorter)fileSorter).SortParts(fileName, "sorted.dat", linesCount, linesCount / 8);

fileSorter.Sort("sorted.dat", out string sortedFileName);

stopWatch.Stop();

Console.WriteLine($"Sorted, file: {sortedFileName}, seconds: {stopWatch.Elapsed.TotalSeconds}");

FileWorker.ShowContent(sortedFileName, 20);

}

}

}

**BinaryFileGenerator.cs:**

namespace lab1 {

public class BinaryFileGenerator : IFileGenerator {

private readonly Random random = new Random();

public void GenerateBySize(string fileName, int megabytes, int minNum = 0, int maxNum = 1000000000) {

using var writer = new BinaryWriter(File.Open(fileName, FileMode.Create));

for (int i = 0; i % 10000 != 0 || !(new FileInfo(fileName).Length >= ByteConverter.MegabytesToBytes(megabytes)); i++) {

writer.Write(random.Next(minNum, maxNum));

}

}

public void GenerateByLinesCount(string fileName, long linesCount, int minNum = 0, int maxNum = 1000000000) {

using var writer = new BinaryWriter(File.Open(fileName, FileMode.Create));

int currentCount = 0;

while (currentCount++ != linesCount) {

writer.Write(random.Next(minNum, maxNum));

}

}

}

}

**BinaryFileSorter.cs:**

namespace lab1 {

public class BinaryFileSorter : IFileSorter {

public void Sort(string fileName, out string sortedFileName, int helpFilesCount = 3) {

if (helpFilesCount < 2)

throw new ArgumentException(null, nameof(helpFilesCount));

string[] BHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"B{i}.dat").ToArray();

string[] CHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"C{i}.dat").ToArray();

SplitFile(fileName, BHelpFiles);

SortHelper(BHelpFiles, CHelpFiles, out sortedFileName);

}

public void SortParts(string fileName, string outputFileName, int size, int shareSize) {

if (File.Exists(outputFileName)) {

File.Delete(outputFileName);

}

int[] array = new int[shareSize];

using var reader = new BinaryReader(File.Open(fileName, FileMode.Open));

using var writer = new BinaryWriter(File.Open(outputFileName, FileMode.OpenOrCreate));

for (int i = 0; i < size / shareSize; i++) {

for (int j = 0; j < shareSize; j++) {

array[j] = reader.ReadInt32();

}

Array.Sort(array);

for (int j = 0; j < shareSize; j++) {

writer.Write(array[j]);

}

}

}

private void SortHelper(string[] BHelpFiles, string[] CHelpFiles, out string fileName) {

var readers = BHelpFiles.Select(f => new BinaryReader(File.OpenRead(f))).ToList();

readers.Where(r => r.EndOfStream()).ToList().ForEach(r => {

r.Dispose();

readers.Remove(r);

});

if (readers.Count == 1) {

fileName = ((FileStream)readers.First().BaseStream).Name;

return;

}

var writers = CHelpFiles.Select(f => new BinaryWriter(File.Open(f, FileMode.Create))).ToList();

var currentWriter = writers.First();

var currentReader = readers.First();

var nums = new List<int>();

var nextNums = new List<int>();

var readerAndPrevNum = readers.ToDictionary(r => r, \_ => int.MinValue);

while (readers.Count != 0) {

while (currentReader.EndOfStream()) {

var readerToRemove = currentReader;

currentReader = readers.NextAfter(currentReader);

readers.Remove(readerToRemove);

readerAndPrevNum.Remove(readerToRemove);

readerToRemove.Dispose();

if (readers.Count == 0) {

break;

}

}

if (readers.Count == 0) {

nums.Sort();

foreach (int n in nums) {

currentWriter.Write(n);

}

currentWriter = writers.NextAfter(currentWriter);

nextNums.Sort();

foreach (int n in nextNums) {

currentWriter.Write(n);

}

break;

}

int num = currentReader.ReadInt32();

if (num >= readerAndPrevNum[currentReader]) {

nums.Add(num);

readerAndPrevNum[currentReader] = num;

}

else {

nextNums.Add(num);

readerAndPrevNum[currentReader] = num;

currentReader = readers.NextAfter(currentReader);

}

if (nextNums.Count >= readers.Count) {

nums.Sort();

foreach (int n in nums) {

currentWriter.Write(n);

}

currentWriter = writers.NextAfter(currentWriter);

nums.Clear();

nums.AddRange(nextNums);

nextNums.Clear();

}

}

readers.ForEach(r => r.Dispose());

writers.ForEach(w => w.Dispose());

SortHelper(CHelpFiles, BHelpFiles, out fileName);

}

private void SplitFile(string fileName, string[] BHelpFiles) {

var writers = BHelpFiles.Select(f => new BinaryWriter(File.Open(f, FileMode.Create))).ToList();

var currentWriter = writers.First();

using var reader = new BinaryReader(File.OpenRead(fileName));

int previousNum = int.MinValue;

while (!reader.EndOfStream()) {

int num = reader.ReadInt32();

if (num >= previousNum) {

currentWriter.Write(num);

}

else {

currentWriter = writers.NextAfter(currentWriter);

currentWriter.Write(num);

}

previousNum = num;

}

writers.ForEach(w => w.Dispose());

}

}

}

**ByteConverter.cs:**

namespace lab1 {

public static class ByteConverter {

public static long MegabytesToBytes(int megabytes) => (long)Math.Pow(2, 20) \* megabytes;

}

}

**Extensions.cs:**

namespace lab1 {

public static class Extensions {

public static T NextAfter<T>(this List<T> source, T item) {

int indexOfItem = source.IndexOf(item);

return indexOfItem < source.Count - 1 ? source[indexOfItem + 1] : source[0];

}

public static bool EndOfStream(this BinaryReader source) {

return source.BaseStream.Position == source.BaseStream.Length;

}

}

}

**FileWorker.cs:**

namespace lab1 {

public class FileWorker {

public static int[] GetArrayPart(int start, int size, string fileName) {

var array = new int[size];

using var reader = new BinaryReader(File.OpenRead(fileName));

for (int i = 0; i < start; i++) {

reader.ReadInt32();

}

for (int i = 0; i < size; i++) {

array[i] = reader.ReadInt32();

}

return array;

}

public static void ShowContent(string fileName, int size) {

var array = GetArrayPart(0, size, fileName);

Console.Write("[ ");

foreach (var i in array) {

Console.Write(i + ", ");

}

Console.WriteLine("]");

}

}

}

**IFileGenerator.cs:**

namespace lab1 {

public interface IFileGenerator {

void GenerateBySize(string fileName, int megabytes, int minNum = 0, int maxNum = 1000000000);

void GenerateByLinesCount(string fileName, long linesCount, int minNum = 0, int maxNum = 1000000000);

}

}

**IFileSorter.cs:**

namespace lab1 {

public interface IFileSorter {

void Sort(string fileName, out string sortedFileName, int helpFilesCount = 3);

}

}

**TextFileGenerator.cs:**

namespace lab1 {

public class TextFileGenerator : IFileGenerator {

private readonly Random random = new Random();

public void GenerateBySize(string fileName, int megabytes, int minNum = 0, int maxNum = 1000000000) {

using var writer = new StreamWriter(fileName, append: false);

for (int i = 0; i % 10000 != 0 || !(new FileInfo(fileName).Length >= ByteConverter.MegabytesToBytes(megabytes)); i++) {

writer.WriteLine(random.Next(minNum, maxNum));

}

}

public void GenerateByLinesCount(string fileName, long linesCount, int minNum = 0, int maxNum = 1000000000) {

using var writer = new StreamWriter(fileName, append: false);

int currentCount = 0;

while (currentCount++ != linesCount) {

writer.WriteLine(random.Next(minNum, maxNum));

}

}

}

}

**TextFileSorter.cs:**

namespace lab1 {

public class TextFileSorter : IFileSorter {

public void Sort(string fileName, out string sortedFileName, int helpFilesCount = 3) {

if (helpFilesCount < 2)

throw new ArgumentException(null, nameof(helpFilesCount));

string[] bHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"B{i}.txt").ToArray();

string[] cHelpFiles = Enumerable.Range(1, helpFilesCount).Select(i => $"C{i}.txt").ToArray();

SplitFile(fileName, bHelpFiles);

SortHelper(bHelpFiles, cHelpFiles, out sortedFileName);

}

private void SortHelper(string[] BHelpFiles, string[] CHelpFiles, out string fileName) {

var readers = BHelpFiles.Select(f => new StreamReader(f)).ToList();

readers.Where(r => r.EndOfStream).ToList().ForEach(r =>

{

r.Dispose();

readers.Remove(r);

});

if (readers.Count == 1) {

fileName = ((FileStream)readers.First().BaseStream).Name;

return;

}

var writers = CHelpFiles.Select(f => new StreamWriter(f, append: false)).ToList();

var currentWriter = writers.First();

var currentReader = readers.First();

var nums = new List<int>();

var nextNums = new List<int>();

var readerAndPrevNum = readers.ToDictionary(r => r, \_ => int.MinValue);

while (readers.Count != 0) {

while (currentReader.EndOfStream) {

var readerToRemove = currentReader;

currentReader = readers.NextAfter(currentReader);

readers.Remove(readerToRemove);

readerAndPrevNum.Remove(readerToRemove);

readerToRemove.Dispose();

if (readers.Count == 0) {

break;

}

}

if (readers.Count == 0) {

nums.Sort();

foreach (int n in nums) {

currentWriter.WriteLine(n);

}

currentWriter = writers.NextAfter(currentWriter);

nextNums.Sort();

foreach (int n in nextNums) {

currentWriter.WriteLine(n);

}

break;

}

int num = int.Parse(currentReader.ReadLine()!);

if (num >= readerAndPrevNum[currentReader]) {

nums.Add(num);

readerAndPrevNum[currentReader] = num;

}

else {

nextNums.Add(num);

readerAndPrevNum[currentReader] = num;

currentReader = readers.NextAfter(currentReader);

}

if (nextNums.Count >= readers.Count) {

nums.Sort();

foreach (int n in nums) {

currentWriter.WriteLine(n);

}

currentWriter = writers.NextAfter(currentWriter);

nums.Clear();

nums.AddRange(nextNums);

nextNums.Clear();

}

}

readers.ForEach(r => r.Dispose());

writers.ForEach(w => w.Dispose());

SortHelper(CHelpFiles, BHelpFiles, out fileName);

}

private void SplitFile(string fileName, string[] BHelpFiles) {

var writers = BHelpFiles.Select(f => new StreamWriter(f, append: false)).ToList();

var currentWriter = writers.First();

using var reader = new StreamReader(fileName);

int previousNum = int.MinValue;

while (!reader.EndOfStream) {

int num = int.Parse(reader.ReadLine()!);

if (num >= previousNum) {

currentWriter.WriteLine(num);

}

else {

currentWriter = writers.NextAfter(currentWriter);

currentWriter.WriteLine(num);

}

previousNum = num;

}

writers.ForEach(w => w.Dispose());

}

}

}

Висновок

При виконанні даної лабораторної роботи мною було вивчено один з алгоритмів зовнішнього сортування під назвою збалансоване багатошляхове злиття.

Спершу розділяємо файл з числами на декілька менших файлів. Сутність роботи алгоритму полягає у знайденні та формуванні серій серед зчитанних елементів новостворених файлів їх злиття та сортування з подальшим перезаписом у наступні файли за кожної ітерації функції, дана процедура буде продовжуватись до моменту остаточного злиття всіх серій у один файл, який буде містити в собі відповідну фінальну відсортовану послідовність.

Було набуто практичних навичок у реалізації алгоритму зовнішнього сортування “Збалансоване багатошляхове злиття”.

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%.