**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-21 Загребельний О.А.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2023

Зміст

[1 Мета лабораторної роботи 3](#_Toc109342184)

[2 ЗаВдання 4](#_Toc109342185)

[3 Виконання 5](#_Toc109342186)

[3.1 Псевдокод алгоритму 5](#_Toc109342187)

[3.2 Програмна реалізація алгоритму 10](#_Toc109342188)

[3.2.1 Вихідний код 10](#_Toc109342189)

[Висновок 25](#_Toc109342190)

[Критерії оцінювання 30](#_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. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

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.

Варіант 12 «Багатофазне сортування»

# Виконання

## Псевдокод алгоритму

PROCEDURE Polyphase (src: Files.File): Files.File;

VAR

i, j, mx, tn: INTEGER;

k, dn, z, level: INTEGER;

x, min: INTEGER;

a, d: ARRAY N OF INTEGER;

t, ta: ARRAY N OF INTEGER;

R: Runs.Rider;

f: ARRAY N OF Files.File;

r: ARRAY N OF Runs.Rider;

PROCEDURE select;

VAR

i, z: INTEGER;

BEGIN

IF d[j] < d[j+1] THEN

INC(j)

ELSE

IF d[j] = 0 THEN

INC(level);

z := a[0];

FOR i := 0 TO N-2 DO

d[i] := z + a[i+1] - a[i];

a[i] := z + a[i+1]

END

END;

j := 0

END;

DEC(d[j])

END select;

PROCEDURE copyrun;

BEGIN

REPEAT

Runs.copy(R, r[j])

UNTIL R.eor

END copyrun;

BEGIN

Runs.Set(R, src);

FOR i := 0 TO N-2 DO

a[i] := 1;

d[i] := 1;

f[i] := Files.New("");

Files.Set(r[i], f[i], 0)

END;

(\*distribute initial runs\*)

level := 1;

j := 0;

a[N-1] := 0;

d[N-1] := 0;

REPEAT

select;

copyrun

UNTIL R.eof OR (j = N-2);

WHILE NOT R.eof DO

select;

IF r[j].first <= R.first THEN

copyrun;

IF R.eof THEN

INC(d[j])

ELSE

copyrun

END

ELSE

copyrun

END

END;

FOR i := 0 TO N-2 DO

t[i] := i;

Runs.Set(r[i], f[i])

END;

t[N-1] := N-1;

REPEAT

z := a[N-2];

d[N-1] := 0;

f[t[N-1]] := Files.New("");

Files.Set(r[t[N-1]], f[t[N-1]], 0);

REPEAT (\*merge one run\*)

k := 0;

FOR i := 0 TO N-2 DO

IF d[i] > 0 THEN

DEC(d[i])

ELSE

ta[k] := t[i];

INC(k)

END

END;

IF k = 0 THEN

INC(d[N-1])

ELSE (\*merge one real run from t[0] ... t[k-1] to t[N-1]\*)

REPEAT

mx := 0;

min := r[ta[0]].first;

i := 1;

WHILE i < k DO

x := r[ta[i]].first;

IF x < min THEN

min := x;

mx := i

END;

INC(i)

END;

Runs.copy(r[ta[mx]], r[t[N-1]]);

IF r[ta[mx]].eor THEN

ta[mx] := ta[k-1];

DEC(k)

END

UNTIL k = 0

END;

DEC(z)

UNTIL z = 0;

Runs.Set(r[t[N-1]], f[t[N-1]]); (\*rotate sequences\*)

tn := t[N-1];

dn := d[N-1];

z := a[N-2];

FOR i := N-1 TO 1 BY -1 DO

t[i] := t[i-1];

d[i] := d[i-1];

a[i] := a[i-1] - z

END;

t[0] := tn;

d[0] := dn;

a[0] := z;

DEC(level)

UNTIL level = 0;

RETURN f[t[0]]

END Polyphase

## Програмна реалізація алгоритму

### Вихідний код

namespace ExternalSort

{

public static class PolyphaseSort

{

private const int tapesCount = 3;

private static int lastTapeIndex = tapesCount - 1;

private const string TEMP\_TAPE = "tape";

public static void Sort(string inputFile, string outputFile)

{

Tape[] tapes = new Tape[tapesCount];

for (int i = 0; i < tapesCount; i++)

{

tapes[i] = new Tape();

}

DistributeSeries(inputFile, tapes);

MergeSeries(outputFile, tapes);

}

private static void DistributeSeries(string inputFile, Tape[] tapes)

{

int fileLength = File.ReadAllLines(inputFile).Length;

StreamReader sr = new StreamReader(inputFile);

for (int i = 0; i < tapesCount; i++)

{

tapes[i].Writer = File.CreateText($"{TEMP\_TAPE}{i}");

}

int chunkLength = Convert.ToInt32(Math.Ceiling(fileLength / Convert.ToDouble(tapesCount - 1)));

for (int i = 0; i < tapesCount - 1; i++)

{

for (int j = 0; j < chunkLength && !sr.EndOfStream; j++)

{

tapes[i].Writer.WriteLine(sr.ReadLine());

}

}

for (int i = 0; i < tapesCount; i++)

{

tapes[i].Writer.Close();

}

sr.Close();

}

private static void MergeSeries(string outputFile, Tape[] tapes)

{

CountSeries(tapes);

AddEmptySeries(tapes);

while (SumOfSeries(tapes) != 1)

{

int minSeries = GetMinSeries(tapes);

int[] numbers = new int[tapesCount - 1];

for (int i = 0, index = 0; i < tapesCount; i++)

{

if (i == lastTapeIndex) continue;

tapes[index].Reader = new StreamReader($"{TEMP\_TAPE}{i}");

if (tapes[i].RealSeries > 0)

{

string line = tapes[index].Reader.ReadLine();

numbers[index] = line != null ? int.Parse(line) : int.MaxValue;

}

else

{

numbers[index] = int.MaxValue;

}

index++;

}

tapes[lastTapeIndex].Writer = new StreamWriter($"{TEMP\_TAPE}{lastTapeIndex}");

int[] last = new int[tapesCount - 1];

for (int i = 0; i < minSeries; i++)

{

while (!MergeFinished(numbers))

{

int minIndex = FindMinIndex(numbers);

tapes[lastTapeIndex].Writer.WriteLine(numbers[minIndex]);

string line = tapes[minIndex].Reader.ReadLine();

int next = line != null ? int.Parse(line) : int.MinValue;

if (next < numbers[minIndex])

{

numbers[minIndex] = int.MaxValue;

last[minIndex] = line != null ? next : int.MaxValue;

}

else

{

numbers[minIndex] = next;

}

}

for (int j = 0; j < tapesCount - 1; j++)

{

numbers[j] = last[j] == 0 ? int.MaxValue : last[j];

}

}

tapes[lastTapeIndex].Writer.Close();

for (int i = 0, index = 0; i < tapesCount; i++)

{

if (i == lastTapeIndex) continue;

StreamWriter temp = new StreamWriter($"temp{i}");

if (tapes[index].Reader.EndOfStream)

{

temp.Close();

index++;

continue;

}

temp.WriteLine(numbers[index]);

while (!tapes[index].Reader.EndOfStream)

{

temp.WriteLine(tapes[index].Reader.ReadLine());

}

temp.Close();

index++;

}

for (int i = 0; i < tapesCount - 1; i++)

{

tapes[i].Reader.Close();

}

for (int i = 0; i < tapesCount; i++)

{

if (i == lastTapeIndex) continue;

File.Delete($"{TEMP\_TAPE}{i}");

File.Move($"temp{i}", $"{TEMP\_TAPE}{i}");

}

bool emptySeriesMerge = true;

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 0) continue;

if (tapes[i].RealSeries >= minSeries)

{

emptySeriesMerge = false;

break;

}

}

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 0)

{

tapes[i].TotalSeries += minSeries;

if (!emptySeriesMerge)

{

tapes[i].RealSeries += minSeries;

}

else

{

tapes[i].EmptySeries += EmptyMergeCount(tapes, minSeries);

tapes[i].RealSeries = tapes[i].TotalSeries - tapes[i].EmptySeries;

}

break;

}

}

for (int i = 0; i < tapesCount; i++)

{

if (i != lastTapeIndex)

{

tapes[i].TotalSeries -= minSeries;

if (tapes[i].RealSeries < minSeries)

{

tapes[i].EmptySeries -= minSeries - tapes[i].RealSeries;

tapes[i].RealSeries = 0;

}

else

{

tapes[i].RealSeries -= minSeries;

}

}

}

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 0)

{

lastTapeIndex = i;

break;

}

}

}

int outputIndex = -1;

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 1)

{

outputIndex = i;

break;

}

}

if (File.Exists(outputFile))

File.Delete(outputFile);

File.Move($"{TEMP\_TAPE}{outputIndex}", outputFile);

for (int i = 0; i < tapesCount; i++)

{

File.Delete($"{TEMP\_TAPE}{i}");

}

}

private static void CountSeries(Tape[] tapes)

{

for (int i = 0; i < tapesCount; i++)

{

using (tapes[i].Reader = new StreamReader($"{TEMP\_TAPE}{i}"))

{

string line;

int subsequenceCount = 0;

int previousNum = int.MinValue; // Initialize with smallest possible value

while ((line = tapes[i].Reader.ReadLine()) != null)

{

if (int.TryParse(line, out int currentNum))

{

// Check if the current number is in non-decreasing order

if (currentNum >= previousNum)

{

previousNum = currentNum;

}

else

{

// Start of a new sorted subsequence

subsequenceCount++;

previousNum = currentNum;

}

}

}

// Increment subsequence count if the last subsequence is not empty

if (previousNum != int.MinValue)

{

subsequenceCount++;

}

tapes[i].RealSeries = subsequenceCount;

tapes[i].TotalSeries = tapes[i].RealSeries;

}

}

}

private static void AddEmptySeries(Tape[] tapes)

{

int maxSeries = GetMaxSeries(tapes);

int[] fibonacciSequence = new int[tapesCount - 1];

for (int i = 0; i < tapesCount - 2; i++)

{

fibonacciSequence[i] = 0;

}

fibonacciSequence[tapesCount - 2] = 1;

int[] totalSeries = new int[tapesCount - 1];

do

{

for (int i = 0, index = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 0) continue;

totalSeries[index] = tapes[i].TotalSeries;

index++;

}

if (maxSeries == fibonacciSequence[tapesCount - 2])

{

if (new HashSet<int>(totalSeries).SetEquals(fibonacciSequence))

{

return;

}

}

int next = fibonacciSequence.Sum();

for (int i = 0; i < tapesCount - 2; i++)

{

fibonacciSequence[i] = fibonacciSequence[i + 1];

}

fibonacciSequence[tapesCount - 2] = next;

} while (!FibonacciEnded(totalSeries, fibonacciSequence));

int[] realSeries = new int[tapesCount];

for (int i = 0; i < tapesCount; i++)

{

realSeries[i] = tapes[i].RealSeries;

}

for (int i = tapesCount - 2; i >= 0; i--)

{

int maxIndex = -1;

int max = 0;

for (int j = 0; j < tapesCount; j++)

{

if (max < realSeries[j])

{

maxIndex = j;

max = realSeries[j];

}

}

realSeries[maxIndex] = 0;

tapes[maxIndex].EmptySeries = fibonacciSequence[i] - tapes[maxIndex].RealSeries;

tapes[maxIndex].TotalSeries = tapes[maxIndex].RealSeries + tapes[maxIndex].EmptySeries;

}

}

public static bool FibonacciEnded(int[] totalSeries, int[] fibonacci)

{

int[] temp = new int[totalSeries.Length];

Array.Copy(totalSeries, temp, totalSeries.Length);

Array.Sort(temp);

for (int i = 0; i < totalSeries.Length; i++)

{

if (temp[i] > fibonacci[i])

{

return false;

}

}

return true;

}

private static int GetMinSeries(Tape[] tapes)

{

int minSeries = int.MaxValue;

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries == 0)

{

continue;

}

if (tapes[i].TotalSeries < minSeries)

{

minSeries = tapes[i].TotalSeries;

}

}

return minSeries;

}

private static int GetMaxSeries(Tape[] tapes)

{

int maxSeries = int.MinValue;

for (int i = 0; i < tapesCount; i++)

{

if (tapes[i].TotalSeries > maxSeries)

{

maxSeries = tapes[i].TotalSeries;

}

}

return maxSeries;

}

private static int FindMinIndex(int[] numbers)

{

int minIndex = -1;

int minValue = int.MaxValue;

for (int i = 0; i < numbers.Length; i++)

{

if (numbers[i] < minValue)

{

minValue = numbers[i];

minIndex = i;

}

}

return minIndex;

}

private static bool MergeFinished(int[] numbers)

{

foreach (int num in numbers)

{

if (num != int.MaxValue)

{

return false;

}

}

return true;

}

private static int SumOfSeries(Tape[] tapes)

{

int sum = 0;

for (int i = 0; i < tapesCount; i++)

{

sum += tapes[i].TotalSeries;

}

return sum;

}

private static int EmptyMergeCount(Tape[] tapes, int minSeries)

{

int maxRealSeries = 0;

for (int i = 0; i < tapes.Length; i++)

{

if (tapes[i].RealSeries > maxRealSeries)

{

maxRealSeries = tapes[i].RealSeries;

}

}

return minSeries - maxRealSeries;

}

private class Tape

{

public int TotalSeries = 0;

public int EmptySeries = 0;

public int RealSeries = 0;

public StreamReader Reader = null!;

public StreamWriter Writer = null!;

public Tape() { }

}

}

}

Висновок

При виконанні даної лабораторної роботи ми вивчили та реалізували алгоритм сортування зовнішніх даних, відомий як "Polyphase Sort". Цей алгоритм розподіляє дані на кілька тимчасових файлів (тейпів) та поступово зливає їх у відсортований результат. Реалізація була проведена за допомогою мови програмування C#.

Алгоритм постійно зчитує та зливає відсортовані серії всередині тейпів, а на останній ітерації залишається одна відсортована послідовність, яка і є вихідним файлом. Проте у великих файлах серій може бути дуже багато, а самі серії можуть бути короткими, що значно збільшує кількість ітерацій в алгоритмі. У результаті сортування проходить довше, ніж могло би, якби серій було менше, а самі серії довшими.

Модифікація алгоритму полягає у тому, щоб перед основним сортуванням записати якомога більше даних з файлу в оперативну пам’ять, відсортувати їх та записати назад у файл. Ця процедура називається PreSort. Таким чином після її роботи у файлі будуть сформовані довгі відсортовані послідовності, що в подальшому зменшить ітерації багатофазного сортування, що у свою чергу значно пришвидшить роботу алгоритму.

Приклади тестування базового алгоритму та модифікованого наведено в таблиці:

|  |  |  |  |
| --- | --- | --- | --- |
| **Time (seconds)** | | **Original** | **Modified** |
|  |
| **File Size (MB)** | **10** | 3.020 | 0.531 |  |
| **50** | 16.568 | 2.323 |  |
| **100** | 46.889 | 5.021 |  |
| **250** | 117.603 | 14.926 |  |
| **512** | 241.962 | 67.078 |  |
| **1024** | 643.029 | 118.171 |  |

Програма дозволяє працювати алгоритму з багатьма проміжними файлами, а саме в діапазоні (3 - 10). Було протестовано швидкість сортування з урахуванням різної кількості проміжних файлів на вхідних даних розміром 10, 50, 100, 250 та 512 мегабайт немодифікованим алгоритмом.

У таблиці наведені результати тестування:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Time (seconds)** | | **File Size (MB)** | | | | |
| **10** | **50** | **100** | **250** | **512** |
| **Number of Tapes** | **3** | 2,948 | 15,888 | 43,829 | 104,583 | 215,234 |
| **4** | 2,379 | 12,773 | 26,08 | 68,442 | 164,17 |
| **5** | 2,203 | 12,014 | 24,368 | 64,205 | 184,299 |
| **6** | 2,155 | 11,105 | 24,355 | 77,986 | 179,151 |
| **7** | 2,167 | 10,741 | 23,657 | 63,703 | 145,576 |
| **8** | 2,207 | 11,011 | 24,318 | 63,157 | 171,195 |
| **9** | 2,115 | 11,383 | 24,856 | 63,11 | 173,635 |
| **10** | 2,188 | 11,458 | 25,405 | 78,032 | 146,977 |
| **Best Time** | | 2,115 | 10,741 | 23,657 | 63,11 | 145,576 |
| **Worst Time** | | 2,948 | 15,888 | 43,829 | 104,583 | 215,234 |
| **Best Tapes Count** | | 9 | 7 | 7 | 9 | 7 |
| **Worst Tapes Count** | | 3 | 3 | 3 | 3 | 3 |

Результати візуалізовано в наступних графіках:

Зображення, що містить текст, знімок екрана, ряд, Графік

Автоматично згенерований опис

Зображення, що містить текст, знімок екрана, ряд, Графік

Автоматично згенерований опис

Зображення, що містить текст, знімок екрана, ряд, Графік

Автоматично згенерований опис

Зображення, що містить текст, знімок екрана, Графік, ряд

Автоматично згенерований опис

Зображення, що містить текст, знімок екрана, Графік, ряд

Автоматично згенерований опис

Зважаючи на результати тестування складно простежити однозначну залежність, проте очевидно, що багатофазне сортування найповільніше працює на трьох файлах, а на більшій кількості до десяти прискорюється. Як показують результати, найчастіше найкращий час сортування показували 7 та 9 проміжних файлів. Таким чином було вирішено спробувати посортувати файл розміром 1ГБ модифікованим алгоритмом на 7 проміжних файлах і отримати близький до мінімального час роботи алгоритму.

Зображення, що містить текст, Шрифт, знімок екрана, типографія

Автоматично згенерований опис

Результат сортування файлу розміром 1ГБ з 7-ма проміжними файлами модифікованим багатофазним алгоритмом.

Критерії оцінювання

У випадку здачі лабораторної роботи до 08.10.2022 включно максимальний бал дорівнює – 5. Після 08.10.2022 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 20%;
* програмна реалізація модифікацій – 20%;
* робота з git – 40%;
* висновок – 5%.