**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”  
Варіант 7

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-15, Гуменюк О.В.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

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 8](#_Toc109342188)

[3.2.1 Вихідний код 8](#_Toc109342189)

[Висновок 16](#_Toc109342190)

[Критерії оцінювання 18](#_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 1Гб за 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

**Function** MultiwayMergeSort(input: Int[], m: Int)

arrsA : Int[1][] = {input}

arrsB : Int[m][]

arrsC : Int[m][]

**for** i = 1 to m **do**

arrsB[i] = Int[]

arrsC[i] = Int[]

**end for**

MultiwayMerge(arrsA, arrsB)

flag : Int = 1

**while** (!isSorted(arrsA[1],arrsB[1], arrsC[1])) **do**

**if** (flag == 1) **do**

MultiwayMerge(arrsB, arrsC)

**end if**

**else do**

MultiwayMerge(arrsC, arrsB)

**end else**

flag = -flag

**end while**

**if** (arrsB[1].length == input.length)

**for** i = 1 to m **do**

input[i] = arrsB[i]

**end for**

**end if**

**else do**

**for** i = 1 to m **do**

input[i] = arrsC[i]

**end for**

**end else**

**return** input

**end**

**Function** isSorted(arrA: Int[], arrB: Int[], arrC: Int[])

**return** (arrB[1].length == arrA.length or

arrC[1].length == arrA.length)

**end**

**Function** MultiwayMerge(inputArrs: Int[][], outputArrs: Int[][])

pointerArr: Int[]

**for** i = 1 to inputArrs.length **do**

pointerArr[i] = 1

**end for**

j : Int = 0

set : Int[]

**while** (!isMerged(pointerArr, inputArrs)) **do**

minIndex : Int = null

minValue : Int = inf

**for** k = 1 to inputArrs.length **do:**

**if** (pointerArr[k] <= inputArrs[k].length) **do**

**if**(set.length == 0 or

inputArrs[k][pointerArr[k]] >= set[set.length]) **do**

**if**(inputArrs[k][pointerArr[k]] <= minValue) **do**

minIndex = k

minValue = inputArrs[k][pointerArr[k]]

**end if**

**end if**

**end if**

**end for**

**if** (minIndex == null) **do**

outputArrs[j+1].append(set)

set.clear()

j = (j+1)% outputArrs.length

**end if**

**else do**

set.append(minValue)

pointerArr[minIndex]++

**end else**

**end while**

**return** outputArrs

**end**

**Function** isMerged(pointerArr: Int[], arrs: Int[][])

**for** i = 1 to arrs.length **do**

**if** (pointerArr[i] <= arrs[i].length) **do**

**return** False

**end if**

**end for**

**return** True

**end**

## Програмна реалізація алгоритму

### Вихідний код неоптимізованого алгоритму

/\*\*

\* Unoptimized program implementation of multiway merge sort

\*/

fun oldMultiwayMergeSort(inputFileName: String, outputFileName: String, m: Int) {

val fileA = arrayOf(File(inputFileName)) //Array with file A

val filesB = Array(m) { i -> File("B${i + 1}.txt").also { it.delete(); it.createNewFile() } } //Array with assisting files B

val filesC = Array(m) { i -> File("C${i + 1}.txt").also { it.delete(); it.createNewFile() } } //Array with assisting files C

multiwayMerge(fileA, filesB) //Before the main loop starts, all elements are transfered from file A to files B

var flag = 1

while (!isSorted(fileA[0], filesB[0], filesC[0])) { // Main loop of the function. Elements are merging from files B

if (flag == 1) { // to files C and back until B1 or C1 is fully sorted

multiwayMerge(filesB, filesC)

} else {

multiwayMerge(filesC, filesB)

}

flag = -flag

}

val outputFile = File(outputFileName) //Destination file

outputFile.createNewFile()

if (filesB[0].length() == fileA[0].length()) {

filesB[0].copyTo(outputFile, true)

} else {

filesC[0].copyTo(outputFile, true)

}

}

/\*\*

\* Checks whether sorting is fully done by comparing length of files B1 and C1 to A

\*/

private fun isSorted(arrA: File, arrB: File, arrC: File): Boolean {

return arrB.length() == arrA.length() || arrC.length() == arrA.length()

}

/\*\*

\* Merges elements from inputFiles to outputFiles

\*/

private fun multiwayMerge(inputFiles: Array<File>, outputFiles: Array<File>) {

val bufferedReaders = Array(inputFiles.size) { i -> BufferedReader(FileReader(inputFiles[i])) } // Array with readers of inputFiles

outputFiles.forEach { it.writeText("")} // Deletes all data from outputFiles before writing to them

var j = 0 // Keeps track in which outputFile to write

val set = ArrayList<Int>(0) // Current set of sorted elements

while (!isMerged(bufferedReaders)) { //Merges files until they are fully merged

var minValue = Int.MAX\_VALUE

var minIndex: Int? = null

for (i in bufferedReaders.indices) { //Find minimum among current elements of all inputFiles

val text = peek(bufferedReaders[i])

if (text != null) { // Checks whether OEF is reached

val num = text.toInt()

if (set.isEmpty() || num >= set.last()) { // Checks whether set is empty or current element is greater than

// it's last element

if (num <= minValue) { // Checks whether current element is smaller than minValue

minValue = num // If all conditions are satisfied: current elements is the new min element

minIndex = i

}

}

}

}

if (minIndex == null) { // If minimum element wasn't found - writes set to the according file

if (outputFiles[j].length() > 0L) { // and empty set

outputFiles[j].appendText("\n")

}

outputFiles[j].appendText(set.joinToString("\n"))

set.clear()

j = (j + 1) % outputFiles.size

} else { // If minimum element was found - just adds it to the set and increase

set.add(minValue) // bufferedReader (file pointer)

bufferedReaders[minIndex].readLine()

}

}

if (outputFiles[j].length() > 0L) { // After main loop ends, program still has to write the last set left

outputFiles[j].appendText("\n")

}

outputFiles[j].appendText(set.joinToString("\n"))

bufferedReaders.forEach { it.close() }

}

/\*\*

\* Checks whether all elements from inputFiles have been merged to outputFiles

\*/

private fun isMerged(readers: Array<BufferedReader>): Boolean {

for (reader in readers) {

val s = peek(reader)

if (s != null) {

return false

}

}

return true

}

/\*\*

\* Gets the next value (line) from bufferedReader and returns to its original position

\*/

private fun peek(reader: BufferedReader): String? {

reader.mark(100)

val line = reader.readLine()

reader.reset()

return line

}

### Тестування неоптимізованого алгоритму

Приклад роботи неоптимізованої версії програми наведений на рисунку 3.1.

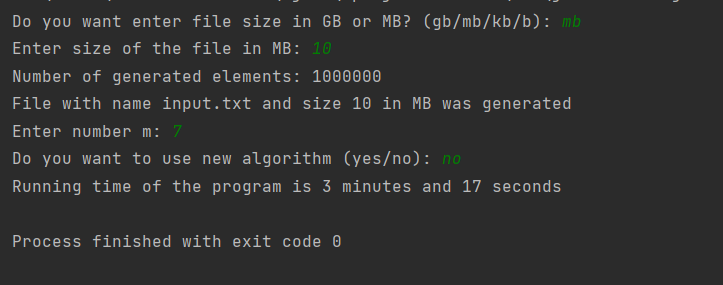


Рисунок 3.1 - Приклад роботи неоптимізованої версії програми

### Вихідний код оптимізованого алгоритму

/\*\*

\* Optimized program implementation of multiway merge sort

\*/

fun newMultiwayMergeSort(inputFileName: String, outputFileName: String, m: Int = 5) {

val fileA = File(inputFileName) //File A

val filesB = Array(m) { i -> File("B${i + 1}.txt").also { it.delete(); it.createNewFile() } } //Array with assisting files B

val filesC = Array(m) { i -> File("C${i + 1}.txt").also { it.delete(); it.createNewFile() } } //Array with assisting files C

initialSorting(fileA, filesB) //Before the main loop starts, chunks of file A (about 250 MB) are sorted and transfered to files B

var flag = 1

while (!isSorted(fileA, filesB[0], filesC[0])) { // Main loop of the function. Elements are merging from files

when (flag){ // to files C and back until B1 or C1 is fully sorted

1 -> multiwayMerge(filesB, filesC)

else -> multiwayMerge(filesC, filesB)

}

flag = -flag

}

val outputFile = File(outputFileName) //Destination file

outputFile.createNewFile()

if (filesB[0].length() == fileA.length()) {

filesB[0].copyTo(outputFile, true)

} else {

filesC[0].copyTo(outputFile, true)

}

}

/\*\*

\* Performs initial soring of 250 MB chunks of file A

\*/

private fun initialSorting(inputFile: File, outputFiles: Array<File>){

val br = BufferedReader(FileReader(inputFile), BUFFER\_SIZE)

val bufferedWritersB = Array(outputFiles.size) {i -> BufferedWriter(FileWriter(outputFiles[i]), BUFFER\_SIZE)}

var i = 0

while (true) {

val charArr = readChunk(br, INITIAL\_CHUNK\_SIZE)

if (charArr.isEmpty()){

break

}

val intArr = charToIntArray(charArr)

quickSort(intArr, 0, intArr.size - 1)

writeToFile(bufferedWritersB[i], outputFiles[i], intArr.joinToString("\n"))

i = (i + 1) % outputFiles.size

}

br.close()

bufferedWritersB.forEach { it.close() }

}

/\*\*

\* Checks whether sorting is fully done by comparing length of files B1 and C1 to A

\*/

private fun isSorted(arrA: File, arrB: File, arrC: File): Boolean {

return arrB.length() == arrA.length() || arrC.length() == arrA.length()

}

/\*\*

\* Merges elements from inputFiles to outputFiles

\*/

private fun multiwayMerge(inputFiles: Array<File>, outputFiles: Array<File>) {

val bufferedWriters = Array(outputFiles.size) { i -> BufferedWriter(FileWriter(outputFiles[i]), BUFFER\_SIZE) } // Array of bufferedReaders of outputFiles

val bufferedReaders = Array(inputFiles.size) { i -> BufferedReader(FileReader(inputFiles[i]), BUFFER\_SIZE) } // Array of bufferedWriters of inputFiles

val outputFilesEmpty = Array(outputFiles.size){true} // Array that keeps track of, which outputFiles are still empty

val bufferedArrays = ArrayList<IntArray>(0) // 2D array that keep current chunks of all inputFiles

val pointerArr = IntArray(inputFiles.size){-1} // Array that keep positions of current elements of bufferedArrays.

// -1 means that the end of array is reached

val chunkSize = CHUNK\_SIZE / inputFiles.size

bufferedReaders.withIndex().forEach{ // Reads first chunks of all non-empty inputFiles

if (peek(it.value) != null){

bufferedArrays.add(charToIntArray(readChunk(bufferedReaders[it.index], chunkSize)))

pointerArr[it.index] = 0

}

}

var j = 0 // Keeps track in which outputFile to write

var set = ArrayList<Int>(0) // Current set of sorted elements

while (!isMerged(pointerArr)) { //Merges files until they are fully merged

val minIndex = findMin(bufferedArrays, pointerArr, set) //Find minimum among current elements of all inputFiles

if (minIndex == null) { // If minimum element wasn't found - writes set to the according file

writeToFile(bufferedWriters[j], !outputFilesEmpty[j], set.joinToString("\n"))

outputFilesEmpty[j] = false

set.clear()

j = (j + 1) % outputFiles.size

}

else if (set.size >= chunkSize/4){ // If set is too big - writes set to the according file, but saves last element

val temp = arrayListOf(set.removeLast()) // last element of set

writeToFile(bufferedWriters[j], !outputFilesEmpty[j], set.joinToString("\n"))

outputFilesEmpty[j] = false

set = temp;

}

if (minIndex != null) { // If minimum element was found - just adds it to the set and increase according pointers

set.add(bufferedArrays[minIndex][pointerArr[minIndex]])

pointerArr[minIndex] ++

if (pointerArr[minIndex] >= bufferedArrays[minIndex].size){ // If the end of the arrays is reached

// reads new chunk and resets the pointer

val s = peek(bufferedReaders[minIndex])

if (s != null) {

bufferedArrays[minIndex] = charToIntArray(readChunk(bufferedReaders[minIndex], chunkSize))

pointerArr[minIndex] = 0

}

else{

pointerArr[minIndex] = -1

bufferedArrays[minIndex] = IntArray(0)

}

}

}

}

writeToFile(bufferedWriters[j], !outputFilesEmpty[j], set.joinToString("\n")) // After main loop ends, program still has to write the last set left

bufferedReaders.forEach { it.close() }

bufferedWriters.forEach { it.close() }

}

/\*\*

\* Writes text in the file and places a "\n" before it, if needed

\*/

private fun writeToFile(writer: BufferedWriter, file: File, text: String){

if (file.length() > 0L) {

writer.write("\n")

}

writer.write(text)

}

/\*\*

\* Writes text in the file and places a "\n" before it, if needed

\*/

private fun writeToFile(writer: BufferedWriter, isEmpty: Boolean, text: String){

if (isEmpty) {

writer.write("\n")

}

writer.write(text)

}

/\*\*

\* Finds minimum element that satisfy all conditions

\*/

private fun findMin(bufferedArrays: ArrayList<IntArray>, pointerArr: IntArray, set: ArrayList<Int>): Int?{

var minValue = Int.MAX\_VALUE

var minIndex: Int? = null

for (i in bufferedArrays.indices) {

if (pointerArr[i] != -1) { // Checks whether end of array is reached

val num = bufferedArrays[i][pointerArr[i]]

if (set.isEmpty() || num >= set.last()) { // Checks whether set is empty or current element is greater than

// it's last element

if (num <= minValue) { // Checks whether current element is smaller than minValue

minValue = num // If all conditions are satisfied: current elements is the new min element

minIndex = i

}

}

}

}

return minIndex

}

/\*\*

\* Checks whether all elements from inputFiles have been merged to outputFiles

\*/

private fun isMerged(pointerArrs: IntArray): Boolean {

for (pointer in pointerArrs) {

if (pointer != -1) {

return false

}

}

return true

}

/\*\*

\* Gets the next value (line) from bufferedReader and returns back to its original position

\*/

private fun peek(reader: BufferedReader): String? {

reader.mark(100)

val line = reader.readLine()

reader.reset()

return line

}

/\*\*

\* Converts charArray to IntArray

\*/

private fun charToIntArray(charArray: CharArray): IntArray {

return charArray.joinToString("").split("\n").map{it.toInt()}.toIntArray()

}

/\*\*

\* Reads and returns chunk of file with specified size

\*/

private fun readChunk(br: BufferedReader, chunkSize: Int): CharArray{

var charArr = CharArray(chunkSize)

br.read(charArr)

if (charArr.last().code != 0){

val arr = ArrayList<Char>()

while(true){

val char = br.read().toChar()

if (char == '\n' || char.code == 0){

charArr += arr

break

}

else{

arr.add(char)

}

}

}

else{

charArr = charArr.filter{it.code != 0}.toCharArray()

}

return charArr

}

### Тестування оптимізованого алгоритму

Приклад роботи оптимізованої версії програми наведений на рисунку 3.2.

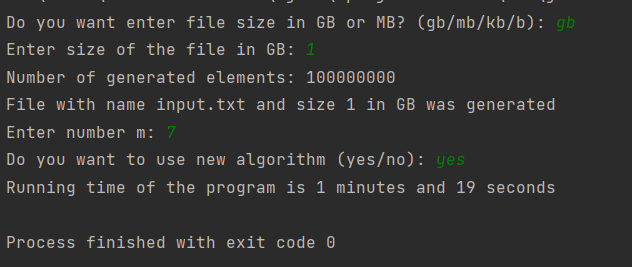


Рисунок 3.2 – Приклад роботи оптимізованої версії програми

Висновок

При виконанні даної лабораторної роботи я отримав практичні навички роботи з алгоритмами зовнішнього сортування, а саме записав псевдокод алгоритму збалансованого багатошляхового злиття, виконав програмну реалізацію цього метода, а також протестував і оптимізував її.

Збалансоване багатошляхове злиття використовує допоміжних файлів (), тому вибір числа m сильно впливає на швидкість роботи алгоритму. Використовуючи тестування було визначено, що для обох версій програмної реалізації алгоритму оптимальним є .

Для оптимізації програмної реалізації алгоритму було виконано дві головні модифікації:

1. попереднє сортування частин файлу;
2. буферизація процесів читання і запису в файли.

У першій версії алгоритму з самого початку відбувається багатошляхове злиття спочатку з файлу А в допоміжні файли , потім з в , потім з цих файлів назад в і так далі, поки в B1 або в C1 не з’явиться повністю відсортовані елементи файлу А. У оптимізованій версії програми перед тим як використати збалансоване багатошляхове злиття, програма читає, зберігає та сортує серії елементів довжиною приблизно 250 МБ. Це значно пришвидшує роботу алгоритму, так як кількість потрібних злиттів сильно зменшується.

У неоптимізованій версії програми майже не використовується буферизація: числа читаються і записуються по-одному. Для кожного окремого запису або читання виходить окреме звертання до файлу, що є набагато повільнішим чим звертання до внутрішньої пам’яті. У оптимізованій версії програми використовуються буфери, а саме буферні читачі (BufferedReader) і буферні записники (BufferedWriter). Також замість читання чисел по-одному, програма читає великі шматки файли, розмір яких залежить від кількості допоміжних файлів. Для всіх буферних виставлений розмір 4096 байтів; збільшення розміру буферів не змінювало швидкість роботи програми.

Критерії оцінювання

У випадку здачі лабораторної роботи до 25.09.2022 включно максимальний бал дорівнює – 5. Після 25.09.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 40%;
* програмна реалізація модифікованого алгоритму – 40%;
* висновок – 5%.