**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»

Варіант - 32

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

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-11 Фукс Вікторія Ігорівна*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc114430853)

[2 Виконання 4](#_Toc114430854)

[2.1 Псевдокод алгоритму 4](#_Toc114430855)

[2.2 Програмна реалізація алгоритму 4](#_Toc114430856)

[2.2.1 Вихідний код 4](#_Toc114430857)

[Висновок 5](#_Toc114430858)

# Мета лабораторної роботи

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

# Виконання

## Псевдокод алгоритму багатофазного сортування

CreateRuns(S)

S размер создаваемых отрезков

CurrentFile=A

while конец входного файла не достигнут do

read S записей из входного файла

sort S записей

if CurrentFile=A then

CurrentFile=B

else

CurrrentFile=A

end if

end while

PolyPhaseMerge(S)

S размер исходных отрезков

Size=S

Input1=A

Input2=B

Current Output=C

while not done do

while отрезки не кончились do

слить отрезок длины Size из файла Input1

с отрезком длины Size из файла Input2

записав результат в CurrentOutput

if (CurrentOutput=A) then

CurrentOutput=B

elsif (CurrentOutput=B) then

CurrrentOutput=A

elsif (CurrentOutput=C) then

Currrent Output=D

elsif (CurrentOutput=D) then

CurrrentOutput=C

end if

end while

Size=Size\*2

if (Input1=A) then

Input1=C

Input2=D

Current Output=A

else

Input1=A

Input2=B

CurrentOutput=C

end if

end while

## Програмна реалізація алгоритму багатофазного сортування

### Вихідний код

Class Main

import java.io.\*;  
import java.util.Arrays;  
import java.util.LinkedList;  
  
public class Main {  
 final static int *sizes* = 1000000;  
  
 public static void main(String[] args) {  
 int runSize = *sizes*;  
 int numFiles = 10;  
 String tempDir = "",  
 outputFileName = "output.txt",  
 inputFileName = "input.txt";  
 boolean gzip = false;  
  
 *inputFileFilling*(inputFileName);  
  
 String[] inputArray = null;  
 try {  
 BufferedReader br = new BufferedReader(new FileReader(inputFileName));  
 inputArray = *readStreamTillEnd*(br);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 System.*err*.println("total lines: " + inputArray.length);  
 System.*err*.println("run size: " + runSize + "\nNum files: " + numFiles + "\nin file: " + inputFileName + "\nout file: " + outputFileName + "\nstdinput: \n");  
  
 Sorter s = new Sorter(runSize, numFiles, tempDir, gzip);  
 s.sort(inputArray, outputFileName);  
 }  
  
 public static void inputFileFilling(String inputFileName) {  
 //sizes  
 int[] inputArray = new int[*sizes*];  
  
 // sizes  
 for (int i = 0; i < *sizes*; i++) {  
 inputArray[i] = (int) (9999+Math.*random*() \*10001);  
 }  
  
 String str = Arrays.*stream*(inputArray)  
 .mapToObj(String::*valueOf*)  
 .reduce((x, y) -> x + "\n" + y)  
 .get();  
  
 try {  
 FileWriter fileWriter = new FileWriter(inputFileName);  
 fileWriter.write(str);  
 fileWriter.close();  
 System.*out*.println("Successfully wrote to the file.\n");  
 } catch (IOException e) {  
 System.*out*.println("An error occurred.");  
 e.printStackTrace();  
 }  
 }  
  
 public static String[] readStreamTillEnd(BufferedReader br) {  
 LinkedList<String> linesList = new LinkedList<String>();  
 try {  
 String line = br.readLine();  
 while (line != null) {  
 linesList.add(line);  
 line = br.readLine();  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 return linesList.toArray(new String[0]);  
 }  
}

Class Sorter

import java.io.\*;  
import java.util.Arrays;  
  
public class Sorter {  
 private boolean DEBUG = true;  
 private StringHeap \_heap;  
 private int \_runs;  
 private int \_maxFiles;  
 private IOManager \_IOMan;  
 private BufferedWriter \_currentStream;  
 private BufferedWriter[] writers;  
 private boolean[] writeIsClosed;  
 private int[] \_fibSequence;  
 private int[] \_currentRuns;  
 private int \_totalRuns = 1;  
 private int \_totalPasses = 0;  
 private String \_outputFile;  
  
 public Sorter(int bufferSize, int maxFiles, String tempDirectory, boolean gzip) {  
 \_heap = new StringHeap(bufferSize);  
 \_maxFiles = maxFiles;  
  
 \_IOMan = new IOManager((gzip ? IOManager.*GZIPPED* : IOManager.*NORMAL*), "UTF-8", 8192, 8192, false, tempDirectory);  
 }  
  
 public void sort(String[] data, String outputFile) {  
  
 \_outputFile = outputFile;  
 \_fibSequence = fibSequence(data.length / (\_heap.capacity() \* 2) + 1, \_maxFiles);  
  
 \_currentRuns = new int[\_maxFiles - 1];  
 \_currentRuns[0] = 1;  
  
 writers = new BufferedWriter[\_maxFiles];  
 writeIsClosed = new boolean[\_maxFiles];  
  
 int i = 0, capacity = \_heap.capacity();  
 for (; i < capacity && i < data.length; i++) {  
 \_heap.insert(data[i]);  
 }  
  
 String biggestInRun;  
  
 for (; i < data.length; i++) {  
  
 biggestInRun = \_heap.peek();  
  
 int compare = data[i].compareTo(biggestInRun);  
 if (compare > 0) {  
 putStreamRuns(\_heap.replace(data[i]));  
 } else if (compare < 0) {  
 putStreamRuns(\_heap.get());  
 \_heap.place(data[i]);  
 } else {  
 putStreamRuns(data[i]);  
 }  
 }  
  
 \_heap = new StringHeap(\_heap.getBase());  
 if (\_heap.size() > 0) {  
 \_runs = runFunction();  
  
 if (writers[\_runs] != null)  
 putStreamRuns("");  
  
 while (\_heap.size() > 0) {  
 putStreamRuns(\_heap.get());  
 }  
 }  
 log("Total Runs: " + \_totalRuns);  
  
 int backup = \_runs;  
  
 while ((\_runs = runFunctionDummy()) != -1) {  
 putStreamRuns("");  
 }  
 \_runs = backup;  
  
 log("Incl. Dummy: " + \_totalRuns);  
  
 for (int g = 0; g < \_maxFiles - 1; g++) {  
 if (writers[g] != null) {  
 try {  
 writers[g].close();  
 writeIsClosed[g] = true;  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 merge();  
 }  
  
 private void deleteFile(String fname) {  
 try {  
 File f = \_IOMan.getFile(fname);  
 if (f.delete()) {  
 debugLog(f.getName() + " is deleted!");  
 } else {  
 debugLog("Delete operation is failed.");  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private int numReaders(BufferedReader[] fr) {  
 int out = 0;  
 for (int i = 0; i < fr.length; i++) {  
 if (fr[i] != null)  
 out++;  
 }  
 return out;  
 }  
  
  
 private void merge() {  
 KVHeap mergeHeap = new KVHeap(\_maxFiles - 1);  
  
 BufferedReader[] fileReaders = new BufferedReader[\_maxFiles];  
  
 int finalRun = -1;  
  
 boolean first = true;  
  
 \_runs = \_maxFiles - 1;  
  
 while (numReaders(fileReaders) > 1 || first) {  
 first = false;  
  
 int tempruns = -1;  
  
 for (int i = 0; i < \_maxFiles; i++) {  
 String inLine = null;  
 try {  
 if (i == \_runs || !\_IOMan.getFile("run" + i).exists())  
 continue;  
  
 if (fileReaders[i] == null)  
 fileReaders[i] = \_IOMan.createBufferedReader("run" + i);  
  
 inLine = fileReaders[i].readLine();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 if (inLine == null) {  
 tempruns = i;  
 try {  
 fileReaders[i].close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 deleteFile("run" + i);  
 fileReaders[i] = null;  
  
 \_totalPasses++;  
 } else if (!inLine.isEmpty()) mergeHeap.insert(new KVPair<Integer, String>(i, inLine));  
  
 }  
  
 if (\_currentStream != null)  
 putStreamMerge("");  
  
 while (mergeHeap.size() > 0) {  
 KVPair<Integer, String> out = mergeHeap.peek();  
 int key = out.key;  
  
 String inLine = null;  
 try {  
 inLine = fileReaders[key].readLine();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 if (inLine == null) {  
 mergeHeap.get();  
 tempruns = key;  
 try {  
 fileReaders[key].close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 deleteFile("run" + key);  
 fileReaders[key] = null;  
  
 \_totalPasses++;  
 } else {  
 if (inLine.isEmpty())  
 mergeHeap.get();  
 else  
 mergeHeap.replace(new KVPair<Integer, String>(key, inLine));  
 }  
  
 putStreamMerge(out.value);  
 }  
  
 if (tempruns != -1) {  
 finalRun = \_runs;  
 \_runs = tempruns;  
 try {  
 \_currentStream.close();  
 \_currentStream = null;  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 log("Total Passes: " + (\_totalPasses - (\_maxFiles - 2)));  
  
 if (\_outputFile != null && new File(\_outputFile).exists())  
 deleteFile(\_outputFile);  
  
 if (\_IOMan.getType() == IOManager.*GZIPPED*) {  
 deleteFile("run" + (finalRun));  
 } else {  
 if (\_outputFile == null) {  
 try {  
 FileInputStream br = new FileInputStream(\_IOMan.getDirectory() + "run" + (finalRun));  
  
 byte[] buffer = new byte[1024];  
 int length;  
 while ((length = br.read(buffer)) > 0) {  
 System.*out*.write(buffer, 0, length);  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.flush();  
 } else {  
 File old = \_IOMan.getFile("run" + finalRun);  
 File output = \_IOMan.getFile(\_outputFile);  
  
 if (old.renameTo(output))  
 debugLog("FINISHED");  
 else  
 debugLog("Error in renaming final output");  
 }  
  
 }  
  
 }  
  
 private int runFunction() {  
 \_totalRuns++;  
 int puttable = (\_runs + 1) % (\_maxFiles - 1);  
 int firstOneChecked = puttable;  
  
 while (\_currentRuns[puttable] >= \_fibSequence[puttable + 1]) {  
 puttable = (puttable + 1) % (\_maxFiles - 1);  
 if (puttable == firstOneChecked) {  
 fibNext(\_fibSequence);  
 }  
 }  
 \_currentRuns[puttable] += 1;  
 return puttable;  
 }  
  
 private int runFunctionDummy() {  
 \_totalRuns++;  
 int puttable = (\_runs + 1) % (\_maxFiles - 1);  
 int firstOneChecked = puttable;  
  
 while (\_currentRuns[puttable] >= \_fibSequence[puttable + 1]) {  
 puttable = (puttable + 1) % (\_maxFiles - 1);  
 if (puttable == firstOneChecked)  
 return -1;  
 }  
  
 \_currentRuns[puttable] += 1;  
 return puttable;  
 }  
  
 private void putStreamRuns(String item) {  
 if (writers[\_runs] == null || writeIsClosed[\_runs]) {  
 writeIsClosed[\_runs] = false;  
 try {  
 writers[\_runs] = \_IOMan.createOutputStreamWriter("run" + \_runs);  
 } catch (Exception e) {  
 debugLog(e.getMessage());  
 }  
 }  
 try {  
 \_IOMan.write(writers[\_runs], item + "\n");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private void putStreamMerge(String item) {  
 if (\_currentStream == null) {  
 try {  
 \_currentStream = \_IOMan.createOutputStreamWriter("run" + \_runs);  
 } catch (Exception e) {  
 debugLog(e.getMessage());  
 }  
 }  
 try {  
 \_IOMan.write(\_currentStream, item + "\n");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private int[] fibNext(int[] previousSequence) {  
 int output = -1;  
 for (int i = 0; i < previousSequence.length; i++) {  
 if (previousSequence[i] == 0)  
 output = i;  
 }  
 if (output == -1)  
 throw new Error("No zeros in sequence? Make sure its a fibonacci sequence");  
  
 for (int i = 0; i < \_maxFiles; i++) {  
 previousSequence[i] += previousSequence[previousSequence.length - 1];  
 }  
  
 previousSequence[previousSequence.length - 1] = 0;  
  
 {  
 Arrays.*sort*(previousSequence);  
 return (previousSequence);  
 }  
  
 }  
  
 private int[] fibSequence(int dataLength, int maxFiles) {  
 int[] returnArray = new int[maxFiles];  
 returnArray[maxFiles - 1] = 1;  
 int output = maxFiles - 1;  
 int total = 1;  
  
 printRuns(returnArray, maxFiles, total);  
 while (true) {  
 for (int i = 0; i < maxFiles; i++) {  
 if (i != output) {  
 returnArray[i] += returnArray[output];  
 total += returnArray[i];  
 }  
 }  
 returnArray[output] = 0;  
 printRuns(returnArray, maxFiles, total);  
 if (total >= dataLength) {  
 Arrays.*sort*(returnArray);  
 return (returnArray);  
 }  
 total = 0;  
 output--;  
  
 if (output < 0) {  
 output += (maxFiles);  
 }  
 }  
 }  
  
 private void printRuns(int[] runsArray, int files, int total) {  
 for (int i = 0; i < files; i++) {  
 debugLog(runsArray[i] + " ");  
 }  
 debugLog("total: " + total + " \n");  
 }  
  
 private void debugLog(String output, Object... args) {  
 if (DEBUG)  
 System.*err*.println(String.*format*(output, args));  
 }  
  
 private void log(String output, Object... args) {  
 System.*err*.println(String.*format*(output, args));  
 }  
}

Class StringHeap

public class StringHeap  
{  
 private int \_size = 0;  
 private String[] \_heapBase;  
  
 public StringHeap(int capacity)  
 {  
 \_heapBase = new String[capacity];  
 }  
  
 public StringHeap(String[] baseArray)  
 {  
 \_heapBase = baseArray;  
  
 for(;\_size < baseArray.length && baseArray[\_size] != null; ++\_size)  
  
 for(int position = (\_size - 1) / 2; position >= 0; position--)  
 {  
 downHeapFrom(position);  
 }  
 }  
  
 public void insert(String item)  
 {  
 \_heapBase[\_size++] = item;  
  
 for(int position = \_size - 1; position > 0;)  
 {  
 int parentPosition = (position - 1) / 2;  
 if(item.compareTo(\_heapBase[parentPosition]) > 0)  
 return;  
  
 swap(position, position = parentPosition);  
 }  
 }  
  
 public String get()  
 {  
 if(\_heapBase[0] == null)  
 return null;  
  
 String smallest = \_heapBase[0];  
  
 \_heapBase[0] = \_heapBase[--\_size];  
  
 downHeapFrom(0);  
 return smallest;  
 }  
  
 public String replace(String item)  
 {  
 String smallest = \_heapBase[0];  
  
 \_heapBase[0] = item;  
 downHeapFrom(0);  
  
 return smallest;  
 }  
  
 public void place(String item)  
 {  
 \_heapBase[\_size] = item;  
 }  
 public String peek()  
 {  
 return \_heapBase[0];  
 }  
 public int size()  
 {  
 return \_size;  
 }  
 public String[] getBase()  
 {  
 return \_heapBase;  
 }  
  
 public int capacity()  
 {  
 return \_heapBase.length;  
 }  
  
 private void downHeapFrom(int fromPosition)  
 {  
 int position = fromPosition;  
  
 for(;;)  
 {  
 int d = position \* 2 + 1;  
 if(d >= \_size)  
 return;  
  
 if(d + 1 < \_size && \_heapBase[d].compareTo(\_heapBase[d + 1]) > 0)  
 d++;  
  
 if(\_heapBase[position].compareTo(\_heapBase[d]) > 0)  
 {  
 swap(position, d);  
 position = d;  
 }  
 else  
 return;  
 }  
 }  
  
 private void swap(int idxA, int idxB)  
 {  
 String tmp = \_heapBase[idxA];  
  
 \_heapBase[idxA] = \_heapBase[idxB];  
 \_heapBase[idxB] = tmp;  
 }  
}

Class KVPair

public class KVPair<K, V> {  
 public K key;  
 public V value;  
  
 public KVPair(K k, V v) {  
 key = k;  
 value = v;  
 }  
}

Class KVHeap

public class KVHeap {  
 private int \_size = 0;  
 private int[] \_heapBaseK;  
 private String[] \_heapBaseV;  
  
 public KVHeap(int capacity) {  
 \_heapBaseK = new int[capacity];  
 \_heapBaseV = new String[capacity];  
 }  
  
 public void insert(KVPair<Integer, String> item) {  
 \_heapBaseK[\_size] = item.key;  
 \_heapBaseV[\_size] = item.value;  
 \_size++;  
  
 for (int position = \_size - 1; position > 0; ) {  
 int parentPosition = (position - 1) / 2;  
 if (item.value.compareTo(\_heapBaseV[parentPosition]) > 0)  
 return;  
  
 swap(position, position = parentPosition);  
 }  
 }  
  
 public KVPair<Integer, String> get() {  
 if (\_heapBaseV[0] == null)  
 return null;  
  
 KVPair<Integer, String> smallest = new KVPair<Integer, String>(\_heapBaseK[0], \_heapBaseV[0]);  
  
 \_heapBaseK[0] = \_heapBaseK[--\_size];  
 \_heapBaseV[0] = \_heapBaseV[\_size];  
  
 downHeapFrom(0);  
 return smallest;  
 }  
  
 public KVPair<Integer, String> replace(KVPair<Integer, String> item) {  
 int smallestK = \_heapBaseK[0];  
 String smallestV = \_heapBaseV[0];  
  
 \_heapBaseK[0] = item.key;  
 \_heapBaseV[0] = item.value;  
 downHeapFrom(0);  
  
 return new KVPair<Integer, String>(smallestK, smallestV);  
 }  
  
 public KVPair<Integer, String> peek() {  
 return new KVPair<Integer, String>(\_heapBaseK[0], \_heapBaseV[0]);  
 }  
  
 public int size() {  
 return \_size;  
 }  
  
 public int capacity() {  
 return \_heapBaseK.length;  
 }  
  
  
 private void downHeapFrom(int fromPosition) {  
 int position = fromPosition;  
  
 for (; ; ) {  
 int d = position \* 2 + 1;  
 if (d >= \_size)  
 return;  
  
 if (d + 1 < \_size && \_heapBaseV[d].compareTo(\_heapBaseV[d + 1]) > 0)  
 d++;  
  
 if (\_heapBaseV[position].compareTo(\_heapBaseV[d]) > 0) {  
 swap(position, d);  
 position = d;  
 } else  
 return;  
 }  
 }  
  
 private void swap(int idxA, int idxB) {  
 int tmpK = \_heapBaseK[idxA];  
 String tmpV = \_heapBaseV[idxA];  
  
 \_heapBaseK[idxA] = \_heapBaseK[idxB];  
 \_heapBaseV[idxA] = \_heapBaseV[idxB];  
  
 \_heapBaseK[idxB] = tmpK;  
 \_heapBaseV[idxB] = tmpV;  
 }  
}

Class IOManager

import java.io.\*;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.zip.GZIPInputStream;  
import java.util.zip.GZIPOutputStream;  
  
public class IOManager {  
 public static int *GZIPPED* = 1;  
 public static int *NORMAL* = 0;  
  
 private int \_type;  
 private String \_charset;  
 private int \_writeBufferSize, \_readBufferSize;  
 private boolean \_autoFlush;  
 private String \_tempDirectory;  
  
 public IOManager(int type, String charset, int writeBufferSize, int readBufferSize, boolean autoFlush, String tempDirectory) {  
 \_type = type;  
 \_charset = charset;  
 \_writeBufferSize = writeBufferSize;  
 \_readBufferSize = readBufferSize;  
 \_autoFlush = autoFlush;  
 \_tempDirectory = tempDirectory.isEmpty() ? "" : tempDirectory + File.*separator*;  
 }  
  
 public BufferedWriter createOutputStreamWriter(String fname) throws Exception {  
 if (\_type == 0)  
 return new BufferedWriter(new OutputStreamWriter(Files.*newOutputStream*(Paths.*get*(\_tempDirectory + fname)), \_charset), \_writeBufferSize);  
 else  
 return new BufferedWriter(new OutputStreamWriter(new GZIPOutputStream(Files.*newOutputStream*(Paths.*get*(\_tempDirectory + fname)), \_writeBufferSize, false), \_charset), \_writeBufferSize);  
 }  
  
 public BufferedReader createBufferedReader(String fname) throws Exception {  
 if (\_type == 0)  
 return new BufferedReader(new InputStreamReader(Files.*newInputStream*(Paths.*get*(\_tempDirectory + fname)), \_charset), \_readBufferSize);  
 else  
 return new BufferedReader(new InputStreamReader(new GZIPInputStream(Files.*newInputStream*(Paths.*get*(\_tempDirectory + fname)), \_readBufferSize), \_charset), \_readBufferSize);  
 }  
  
 public void write(BufferedWriter writer, String item) throws IOException {  
 writer.write(item);  
 if (\_autoFlush)  
 writer.flush();  
 }  
  
 public File getFile(String fname) {  
 return new File(\_tempDirectory + fname);  
 }  
  
 public int getType() {  
 return \_type;  
 }  
  
 public String getDirectory() {  
 return \_tempDirectory;  
 }  
}

Висновок

При виконанні даної лабораторної роботи ми вивчили основні алгоритми зовнішнього сортування та способи їх модифікації, оцінили поріг їх ефективності. Розробили програму на мові Java та написали псевдокод.