**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-15 Мочалов Дмитро Юрійович*

**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 14](#_Toc109342190)

[Критерії оцінювання 15](#_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

func MultiwayMergeSort(string[] fileNameList, int chunk):

    int check = 1

    while(!isSorted((fileB1 or fileC1) from fileNameList)) do:

      Queue[] listFromFiles = ReadFiles(int check, string[] fileNameList)

      Merge(listFromFiles)

      check = check \* (-1)

    end while

end func

func Merge(int check, int chunk, Queue[] listFromFiles):

  long[] series

  string fileName

  if check > 0:

    fileName = "C"

  else:

    fileName = "B"

  end if

  int fileNumber = 1

  while(!isEmpty(listFromFiles)) do:

    long minValue = mav value of long

    int? minIndex = null

    for i = 0 to listFromFiles.Length do:

      if listFromFiles[i].Length != 0 do:

        if (series.Length == 0 or listFromFiles[i].Peek >= series.LastElement) and listFromFiles[i].Peek <= minValue do:

          minValue = listFromFiles[i].Peek

          minIndex = i

        end if

      end if

    end for

    if(minIndex == null) do :

     WriteSeriesToFile(filename + fileNumber)

     series.clear

     fileNumber = fileNumber % chunk + 1

    else do :

      series.Add(list[minIndex].Dequeu)

    end if

  end while

end func

func isSorted(listFromFiles) do:

  long[] ArrA = listFromFiles.IndexOf(fileA)

  long[] ArrB1 = listFromFiles.IndexOf(fileb1)

  long[] ArrC1 = listFromFiles.IndexOf(filec1)

  if ArrA.Length == ArrB1.Length or ArrA.Length == ArrC1.Length:

   return true

  end if

  return false

func isEmpty(Queue[] list) do:

  int count = 0

  foreach element in list:

    count += element.Length

  end foreach

  if length !=0 do:

   return false

  end if

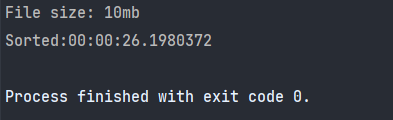
  return true

## Програмна реалізація алгоритму

### Вихідний код

*namespace* Lab1;  
  
*public class* SimpleMultiwaySort  
{  
 *public string* PathToAFolder { get; }  
 *public int* Size { get; set; }  
 *public* List<*string*> Filelist { get; set; }  
   
  
 *public* SimpleMultiwaySort(*string* path, *int* size)  
 {  
 PathToAFolder = path;  
 Size = size;  
 Filelist = *new* List<*string*>();  
 }  
 *public void* GenerateFile()  
 {  
 Random random = *new* Random();  
 *using* BinaryWriter binaryWriter = *new* BinaryWriter(File.Open(PathToAFolder + "A.bin" , FileMode.Create));  
 *for* (*int* i = 0; i < Size/8; i++)  
 {  
 binaryWriter.Write(random.NextInt64(1,1000000));  
 }  
 }  
  
 *private* Queue<*long*> ReadFromInputFile()  
 {  
 Queue<*long*> arrFromInputFile = *new* Queue<*long*>();  
 *using* (BinaryReader binaryReader = *new* BinaryReader(File.Open(PathToAFolder + "A.bin", FileMode.Open)))  
 {  
 *while* (binaryReader.BaseStream.Position != binaryReader.BaseStream.Length)  
 {  
 arrFromInputFile.Enqueue(binaryReader.ReadInt64());  
 }  
 }  
  
 *return* arrFromInputFile;  
 }  
 *public void* SplitFile(*int* n)  
 {  
 Queue<*long*> arrFromInputFile = ReadFromInputFile();  
 List<Queue<*long*>> listInFiles = *new* List<Queue<*long*>>(n);  
 *for* (*int* i = 0; i < n; i++)  
 {  
 listInFiles.Add(*new* Queue<*long*>());  
 }  
 *int* j = 1;  
 *foreach* (*var* elem *in* arrFromInputFile)  
 {  
 listInFiles[j-1].Enqueue(elem);  
 j = j % n + 1;  
 }  
 WriteInOutputFiles(listInFiles,n);  
   
 }  
  
 *private void* WriteInOutputFiles(List<Queue<*long*>> arr, *int* n)  
 {  
 *for* (*int* i = 0; i < n; i++)  
 {  
 *using* (BinaryWriter binaryWriter = *new* BinaryWriter(File.Open(PathToAFolder + $"B{i+1}.bin", FileMode.Create)) )  
 {  
 *foreach* (*var* ele *in* arr[i])  
 {  
 binaryWriter.Write(ele);  
 }  
 Filelist.Add($"B{i+1}.bin");  
 }  
 }  
  
 *for* (*int* i = 0; i < n; i++)  
 {  
 *using* (BinaryWriter binaryWriter = *new* BinaryWriter(File.Open(PathToAFolder + $"C{i+1}.bin", FileMode.Create)) )  
 {  
 Filelist.Add($"C{i+1}.bin");  
 }  
 }  
 }  
   
 *private* List<Queue<*long*>> ReadFiles(*int* check,*int* n)  
 {  
 List<Queue<*long*>> listsFromFiles = *new* List<Queue<*long*>>(n);  
 *if* (check>0)  
 {  
 *for* (*int* i = 0; i < n; i++)  
 {  
 *using* (BinaryReader binaryReader = *new* BinaryReader(File.Open(PathToAFolder + $"{Filelist[i]}", FileMode.Open)) )  
 {  
 *if* (binaryReader.BaseStream.Length != 0)  
 {  
 *var* tmpQueue = *new* Queue<*long*>();  
 *while* (binaryReader.BaseStream.Position != binaryReader.BaseStream.Length)  
 {  
 tmpQueue.Enqueue(binaryReader.ReadInt64());  
 }  
 listsFromFiles.Add(tmpQueue);  
 }  
 }  
 }  
 }  
 *else* {  
 *for* (*int* i = n; i < 2\*n; i++)  
 {  
 *using* (BinaryReader binaryReader = *new* BinaryReader(File.Open(PathToAFolder + $"{Filelist[i]}", FileMode.Open)) )  
 {  
 *if* (binaryReader.BaseStream.Length != 0)  
 {  
 *var* tmpQueue = *new* Queue<*long*>();  
 *while* (binaryReader.BaseStream.Position != binaryReader.BaseStream.Length)  
 {  
 tmpQueue.Enqueue(binaryReader.ReadInt64());  
 }  
 listsFromFiles.Add(tmpQueue);  
 }  
 }  
 }  
 }  
 *return* listsFromFiles;  
 }  
  
 *public void* Sort(*int* n)  
 {  
 *int* check = 1;  
 *while* (*true*)  
 {  
 FileInfo[] files = {  
 *new* FileInfo(PathToAFolder + Filelist[0]),  
 *new* FileInfo(PathToAFolder + Filelist[n])  
 };  
 *// ReSharper disable once ComplexConditionExpression  
 if* (Size == files[0].Length || Size == files[1].Length) *break*;  
  
 List<Queue<*long*>> listsFromFiles = ReadFiles(check,n);  
 *// if (inputArr.Count == listsFromFiles[0].Count) break;* Merge(check,n,listsFromFiles);  
 check \*= -1;  
 }  
 }  
  
 *private void* Merge(*int* check,*int* n, List<Queue<*long*>> list)  
 {  
 List<*long*> series = *new* List<*long*>();  
 *string* fileName;  
 *if* (check > 0)  
 {  
 fileName = "C";  
 }  
 *else* fileName = "B";  
 *int* fileNumber = 1;  
 *while* (!isEmpty(list))  
 {  
 *var* minValue = *long*.MaxValue;  
 *int*? minIndex = *null*;  
 *//searching minimal element from series  
 for* (*int* i = 0; i < list.Count; i++)  
 {  
 *if* (list[i].Count != 0)  
 {  
 *long* tmp = list[i].Peek();  
 *if* (series.Count == 0 || tmp >= series.Last()) *// check if series is empty, then first minimal will be added to series else check if current element is bigger than last elements in series and lower than current minimal* {  
 *if* (tmp <= minValue)  
 {  
 minValue = tmp;  
 minIndex = i;  
 }  
 }  
 }   
 }  
 *//if minimal isn't found, than write series to file and clear it  
 if* (minIndex == *null*)  
 {  
 *using* (BinaryWriter binaryWriter = *new* BinaryWriter(File.Open(PathToAFolder +$"{fileName}{fileNumber}.bin",FileMode.Append)) )  
 {  
 *foreach* (*var* ele *in* series)  
 {   
 binaryWriter.Write(ele);  
 }  
 }  
 fileNumber = fileNumber % n + 1;  
 series.Clear();  
 }  
 *//else add minimum to series  
 else* {  
 series.Add(list[(*int*)minIndex].Dequeue());  
 }  
 }  
 *// add to file last found series  
 using* (BinaryWriter binaryWriter = *new* BinaryWriter(File.Open(PathToAFolder +$"{fileName}{fileNumber}.bin",FileMode.Append)) )  
 {  
 *foreach* (*var* ele *in* series)  
 {   
 binaryWriter.Write(ele);  
 }  
 }  
 *//clear input files* ClearFiles(check,n);  
 }  
 *private void* ClearFiles(*int* check,*int* n)  
 {  
 *string* fileName;  
 *if* (check > 0)  
 {  
 fileName = "B";  
 }  
 *else* {  
 fileName = "C";  
 }   
 *for* (*int* i = 0; i < n; i++)  
 {  
 FileStream fileStream = File.Open(PathToAFolder + $"{fileName}{i+1}.bin", FileMode.Open);  
 fileStream.SetLength(0);  
 fileStream.Close();  
 }  
 }  
 *private bool* isEmpty(List<Queue<*long*>> list)  
 {  
 *int* count = 0;  
 *foreach* (*var* queue *in* list)  
 {  
 count += queue.Count;  
 }  
 *if* (count != 0)  
 {  
 *return false*;  
 }  
 *return true*;  
 }  
  
 *public void* OutPut()  
 {  
 *foreach* (*var* file *in* Filelist)  
 {  
 *int* countOfElements = 0;  
 *using* (BinaryReader binaryReader = *new* BinaryReader(File.Open(PathToAFolder + file, FileMode.Open)))  
 {  
 *if* (binaryReader.BaseStream.Length != 0)  
 {  
 *while* (binaryReader.BaseStream.Position != binaryReader.BaseStream.Length)  
 {  
 countOfElements++;  
 Console.WriteLine(binaryReader.ReadInt64());  
 }  
  
 Console.WriteLine("Count of elements:" + countOfElements);  
 }  
 }  
  
 }  
 }  
}

### Результати алгоритму:



Висновок

При виконанні даної лабораторної роботи я вивчив алгоритм зовнішнього сортування збалансоване багатошляхове злиття, який базуєтся на розділенні вхідних данних на серії і їх розподіл по допоміжним файлам Вn. Потім їх злиття в файли Сn і так далі, поки все не зіллєтся в один файл у відсортованому вигляді.

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%.