**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 Мельник М.О.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc115541724)

[2 Завдання 4](#_Toc115541725)

[3 Виконання 6](#_Toc115541726)

[3.1 Псевдокод алгоритму 6](#_Toc115541727)

[3.2 Програмна реалізація алгоритму 9](#_Toc115541728)

[3.2.1 Вихідний код 9](#_Toc115541729)

[3.3 Тестування програмної реалізації 12](#_Toc115541730)

[4 Модифікація 13](#_Toc115541731)

[4.1 Теоретичні відомості 13](#_Toc115541732)

[4.2 Програмна реалізація 13](#_Toc115541733)

[4.2.1 Вихідний код 13](#_Toc115541734)

[4.3 Тестування 15](#_Toc115541735)

[Висновок 16](#_Toc115541736)

[Критерії оцінювання 17](#_Toc115541737)

# 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

Варіант 18

Природне (адаптивне) злиття

## Псевдокод алгоритму

PROCEDURE Sort()

WHILE NOT IsSorted(path\_a) DO

Distribute()

Merge()

END PROCEDURE Sort()

PROCEDURE IsSorted(path\_a)

a = open(path\_a, mode=”rb”)

curr = a.read()

next = a.read()

WHILE next DO

IF curr > next THEN

a.close()

RETURN FALSE

curr = next

next = a.read()

a.close()

RETURN TRUE

END PROCEDURE IsSorted()

Розподіл між файлами, достатньо банально зчитуємо поступово файл А, дописуємо цей елементвідповідно до файлу В чи С в залежності від прапора і, якщо ж це кінець серії, то змінюємо прапор на протилежний

PROCEDURE Distribute(path\_a, path\_b, path\_c)

a = open(path\_a, mode=”RB”)

b = open(path\_b, mode=”WB”)

c = open(path\_c, mode=”WB”)

i = True

curr = a.read()

next = a.read()

WHILE curr DO

IF i THEN

b.write(curr)

ELSE

c.write(curr)

IF curr > next THEN

i = NOT i

curr = next

next = a.read()

a.close()

b.close()

c.close()

END PROCEDURE Distribute()

*Процедура злиття. Якщо обидва дані числа не в кінці серії, ми знаходимо менше, його дописуємо, і посуваємо покажчик, таким чином зливаючи ці дві серії*

*Інакше якщо число файлу В в кінці серії, то ми дозливаємо актуальну cерію файла В і залишки серії файла С, тобто якщо число із файла С менше, ми його дописуємо, як тільки меншим виявиться оце наше останнє число серії із файла В, то ми це число дописуємо, і всі інші числа із серії із файла С так само дописуємо аж до кінця серії. Після цієї операції обидва покажчика будуть на початку нової серії*

*Аналогічно робимо якщо число файлу С в кінці серії*

*Якщо ж обидва числа в кінці серії, дописуємо їх по зростанню і суваємо покажчик на крок перед, в результаті обидва покажчики на початку серії*

*Якщо в кінці фалу, дописуємо залишок іншого файлу*

PROCEDURE Merge(path\_a, path\_b, path\_c)

a = open(path\_a, mode=”WB”)

b = open(path\_b, mode=”RB”)

c = open(path\_c, mode=”RB”)

curr\_b = b.read()

curr\_c = c.read()

next\_b = b.read()

next\_c = c.read()

WHILE curr\_b AND curr\_c DO

IF curr\_b <= next\_b AND curr\_c <= next\_c THEN

IF curr\_b <= curr\_c THEN

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

ELSE

a.write(current\_c)

curr\_c = next\_c

next\_c = c.read()

ELSE IF curr\_b >= next\_b AND curr\_c <= next\_c THEN

WHILE curr\_c <= next\_c DO

IF curr\_b <= curr\_c THEN

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

WHILE curr\_c <= next\_c DO

a.write(curr\_c)

curr\_c = next\_c

next\_c = c.read()

a.write(curr\_c)

curr\_c = next\_c

next\_c = c.read()

BREAK

ELSE

a.write(curr\_c)

curr\_c = next\_c

next\_c = c.read()

ELSE IF curr\_c >= next\_c AND curr\_b <= next\_b THEN

WHILE curr\_b <= next\_b DO

IF curr\_c <= curr\_b THEN

a.write(curr\_c)

curr\_c = next\_c

next\_c = c.read()

WHILE curr\_b <= next\_b DO

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

BREAK

ELSE

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

ELSE

IF curr\_c <= curr\_b THEN

a.write(curr\_c)

a.write(curr\_b)

ELSE

a.write(curr\_b)

a.write(curr\_c)

curr\_c = next\_c

curr\_b = next\_b

next\_c = c.read()

next\_b = b.read()

if NOT curr\_b and curr\_c THEN

WHILE curr\_c DO

a.wriet(curr\_c)

curr\_c = next\_c

next\_c = c.read()

ELSE IF NOT curr\_c and b\_curr THEN

WHILE curr\_b DO

a.write(curr\_b)

curr\_b = next\_b

next\_b = b.read()

a.close()

b.close()

c.close()

END PROCEDURE Merge()

## Програмна реалізація алгоритму

### Вихідний код

from random import randint

import shutil

class Reader:

"""Class representing a binary file and two reading pointers: current and next"""

def \_\_init\_\_(self, path: str):

self.path = path

self.f = open(path, "rb")

self.curr = self.f.read(32)

self.next = self.f.read(32)

def close(self):

self.f.close()

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

"""Return current pointer and moves pointers forward by 1"""

tmp = self.curr

self.curr = self.next

self.next = self.f.read(32)

return tmp

class Sorter:

def \_\_init\_\_(self, path\_a: str, path\_b: str, path\_c: str):

"""File B and C is cleared, file A is not"""

self.path\_a = path\_a

self.path\_b = path\_b

self.path\_c = path\_c

self.\_clear(path\_b)

self.\_clear(path\_c)

def create\_random\_file(self, n: int, max\_n: int):

with open(self.path\_a, "wb") as a:

for i in range(n):

a.write(randint(1, max\_n).to\_bytes(32, byteorder="big"))

def copy\_from\_file(self, path: str):

shutil.copy(path, self.path\_a)

def sort(self):

while not self.\_is\_sorted():

self.\_distribute()

self.\_merge()

def \_distribute(self):

a = Reader(self.path\_a)

b = open(self.path\_b, "wb")

c = open(self.path\_c, "wb")

i = True

while a.curr:

b.write(a.curr) if i else c.write(a.curr)

if a.curr > a.next:

i = not i

next(a)

a.close(), b.close(), c.close()

def \_merge(self):

"""

Злиття відповідних серій

Логіка наступна:

Якщо обидва дані числа не в кінці серії, ми знаходимо менше, його дописуємо, і посуваємо

покажчик, таким чином зливаючи ці дві серії

Інакше якщо число файлу В в кінці серії, то ми дозливаємо актуальну серію файла В і залишки

серії файла С, тобто якщо число із файла С менше, ми його дописуємо, як тільки меншим

виявиться оце наше останнє число серії із файла В, то ми це число дописуємо, і всі інші

числа із серії із файла С так само дописуємо аж до кінця серії. Після цієї операції обидва

покажчика будуть на початку нової серії

Аналогічно робимо якщо число файлу С в кінці серії

Якщо ж обидва числа в кінці серії, дописуємо їх по зростанню і суваємо покажчик на крок уперед,

в результаті обидва покажчики на початку серії

Якщо в кінці фалу, дописуємо залишок іншого файлу

"""

a = open(self.path\_a, "wb")

b = Reader(self.path\_b)

c = Reader(self.path\_c)

while b.curr and c.curr:

if b.curr <= b.next and c.curr <= c.next:

if b.curr <= c.curr:

a.write(next(b))

else:

a.write(next(c))

elif b.curr >= b.next and c.curr <= c.next:

while c.curr <= c.next:

if b.curr <= c.curr:

a.write(next(b))

while c.curr <= c.next:

a.write(next(c))

a.write(next(c))

break

else:

a.write(next(c))

elif c.curr >= c.next and b.curr <= b.next:

while b.curr <= b.next:

if c.curr <= b.curr:

a.write(next(c))

while b.curr <= b.next:

a.write(next(b))

a.write(next(b))

break

else:

a.write(next(b))

else:

if c.curr <= b.curr:

a.write(c.curr)

a.write(b.curr)

else:

a.write(b.curr)

a.write(c.curr)

next(c), next(b)

if not b.curr and c.curr:

while c.curr:

a.write(next(c))

elif not c.curr and b.curr:

while b.curr:

a.write(next(b))

a.close(), b.close(), c.close()

def \_is\_sorted(self):

"""Check if file A is already sorted"""

a = Reader(self.path\_a)

while a.next:

if a.curr > a.next:

a.close()

return False

next(a)

a.close()

return True

@staticmethod

def \_clear(path: str):

with open(path, "wb"):

pass

## Тестування програмної реалізації

Для тестування створимо файл розміром в 32 Мб за допомогою методу create\_random\_file(1028\*\*2, 1\_000\_00) та здійснимо його сортування, виведемо перші 30 чисел файлу:

sorter = Sorter("a.bin", "b.bin", "c.bin")

sorter.create\_random\_file(1024\*\*2, 1\_000\_000)

sorter.sort()

print(sorter)

Отримали:

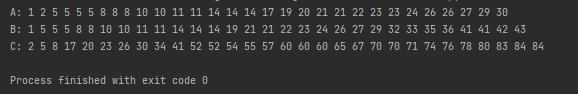


Рисунок 3.1 — тестування програмної реалізації

# Модифікація

## Теоретичні відомості

Для пришвидшення швидкодії сортування використаємо наступні методи:

1. Етап підготовки, у якому файл зчитується чанками обсягом, які поміщаються у ОП, у буфер і сортуються будь-яким алгоритмом внутрішнього сортування, у нашому випадку вбудованим у Python Timsort, після чого записуються у файл. В результаті початковий файл А складатиметься із серій довжиною n крім, можливо, останньої.
2. Оскільки ми знаємо, що початковий файл складається із серій довжиною n, а після кожного етапу сортування — із серій довжиною 2in, де i — ітерація, під час етапу розподілу ми можемо використати цю інформацію наступним чином: під час першої ітерації ми зчитуємо чанк розміром n, і переписуємо його у файл B, наступний чанк ми переписуємо у файл C і так далі. На другій ітерації — перші два чанки записуємо у файл B, наступні 2 — у файл С, потім по 4, потім по 8 чанків і так далі.
3. Знаючи розмір файлу, можна з легкістю визначити, скільки знадобиться ітерацій сортування, а саме , де N — обсяг початкового файла.

## Програмна реалізація

Створимо клас BufferedSorter, який буде підкласом Sorter:

### Вихідний код

from sorter import Sorter

import os

import math

class BufferedSorter(Sorter):

def \_\_init\_\_(self, path\_a: str, path\_b: str, path\_c: str, n: int):

super().\_\_init\_\_(path\_a, path\_b, path\_c)

self.iteration = 0

self.split\_n = 1

self.n = n

try:

self.iter\_max = int(int(math.log2(os.stat(self.path\_a).st\_size // 32 // self.n))) + 1

except ValueError:

self.iter\_max = 1

def create\_random\_file(self, n: int, max\_n: int):

super().create\_random\_file(n, max\_n)

self.iter\_max = int(int(math.log2(os.stat(self.path\_a).st\_size // 32 // self.n))) + 1

def copy\_from\_file(self, path: str):

super().copy\_from\_file(path)

self.iter\_max = int(int(math.log2(os.stat(self.path\_a).st\_size // 32 // self.n))) + 1

def buffered\_sort(self):

"""Around 2 \* 32 \* n of RAM used max while buffering"""

self.\_buffered\_prepare()

for \_ in range(self.iter\_max):

self.\_buffered\_distribute()

self.\_merge()

self.iteration += 1

def \_buffered\_prepare(self):

"""Read chunks from file A, write sorted chunks into temporary file, copy it to file A"""

tmp = open("tmp", "wb")

a = open(self.path\_a, "rb")

buffer = []

for \_ in range(self.n):

x = a.read(32)

if not x:

break

buffer += x,

while buffer:

buffer.sort()

for x in buffer:

tmp.write(x)

del buffer

buffer = []

for \_ in range(self.n):

x = a.read(32)

if not x:

break

buffer += x,

tmp.close()

a.close()

self.copy\_from\_file("tmp")

os.remove("tmp")

def \_buffered\_distribute(self):

a = open(self.path\_a, "rb")

b = open(self.path\_b, "wb")

c = open(self.path\_c, "wb")

buffer = []

for \_ in range(self.n):

x = a.read(32)

if not x:

break

buffer += x,

i = True

j = 2

while buffer:

for x in buffer:

b.write(x) if i else c.write(x)

if j > self.split\_n:

i = not i

j = 1

j += 1

del buffer

buffer = []

for \_ in range(self.n):

x = a.read(32)

if not x:

break

buffer += x,

self.split\_n \*= 2

a.close(), b.close(), c.close()

## Тестування

Для тестування було створено файл “32Gb.bin” із масивом випадково згенерованих чисел розміром 32 Гб для машини із обсягом ОП в 16 Гб. Для тестування було обрано розбиття на чанки розмірами 4 Гб, тобто 25% від ОП машини.

В результаті сортування було здійснене за 111 хв, або приблизно 3,47 хв/Гб, що є непоганим результатом враховуючи особливості машини, мови програмування і той факт, що в кожен момент часу на зберігання даних було задіяно всього 25% від загального обсягу ОП.

Висновок

При виконанні даної лабораторної роботи було розроблено алгоритм реалізації зозвнішнього адаптивного злиття у вигляді псевдокоду, була здійснена його реалізація на Python та тестування на випадково згенерованому файлі розміру 32 Мб.

Була здійснена модифікація програми, а саме впорядкування елементів перед сортуванням у серії довжини n з використанням внутрішнього сортування; покращена операція розподілу, у якій враховується попереднє впорядкування елементів і не здійснюється визначення серій, натомість здійснюється, в залежності від ітерації, запис у файли B та C серій довжиною, кратною n; автоматичне визначення кількості ітерацій, що знадобляться, перед початком сортування щоб не перевіряти файл А на відсортованість кожної ітерації.

Модифікація дозволила значно пришвидшити алгоритм за рахунок використання внутрішнього сортування наскільки можливо, значного зменшення кількості ітерацій шляхом зменшення кількості серій у початковому файлі, відсутності читання файлу А перед кожною ітерацією сортування та більш швидкої реалізацієї операції розподілу. У свою чергу оригінальний варіант алгоритму дозволяє доволі консервативно використовувати ОП.

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%.