**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 Орищенко Я.О.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

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 6](#_Toc109342188)

[3.2.1 Вихідний код 6](#_Toc109342189)

[Висновок 7](#_Toc109342190)

[Критерії оцінювання 8](#_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 sort():**

find\_series()

**WHILE** length(“file.bin”) != length(“B1.bin”) and length(“file.bin”) != length(“C1.bin”) **DO**:

merge()

**func find\_series()**

a = open(“file.bin”, mode=“read”)

**FOR** i = 0, i < files\_amount:

file[i] = open(“B{i+1}.bin”, mode=”write”)

series = [a.curr]

file\_number = 0

**FOR** i=0, i < size(a) **DO**:

**IF** a.next >= a.curr **DO**:

series += a.next

**ELSE**:

**FOR** j = 0, j < length(series) **DO**:

file[file\_number].write(series[j])

file\_number += 1

**IF** file\_number == files\_amount **DO**:

file\_number = 0

series = [a.next]

next(a)

**func merge(file\_name1, file\_name2):**

merge\_files = []

**FOR** i = 0; i<files\_amount **DO**:

clear\_file(“file\_name2{i+1}.bin”)

merge\_files[i] = open(“file\_name2{i+1}.bin”, mode=”append”)

distribute\_files = []

**FOR** i = 0; i<files\_amount **DO**:

clear\_file(“file\_name1{i+1}.bin”)

distribute\_file [i] = open(“file\_name1{i+1}.bin”, mode=”read”)

files\_read = False

**WHILE** files\_read == False **DO**:

**FOR** i=0, i<length(merge\_files) **DO**:

**IF** all\_files\_read(distribute\_files) **DO**:

break

**FOR** j=0; j<length(distribute\_files) **DO**:

distribute\_files[j].prev = 0

**WHILE** True **DO**:

min\_elem\_file = -1

min\_elem = 10001

**FOR** j=0; j<length(distribute\_files) **DO**:

**IF** distribute\_files[j].prev <= distribute\_files[j].curr <= min\_elem:

min\_elem = distribute\_files[j].curr

min\_elem\_file = j

**IF** min\_elem == 10001:

break

merge\_files[i].write(min\_elem)

next(distribute\_files[min\_elem\_file])

## Програмна реалізація алгоритму

### Вихідний код

**main.py**

import time  
from file\_manager import Sort, AdvancedSort  
  
size = 1024  
file = Sort("file.bin", size, 16)  
file.create\_file()  
start = time.time()  
file.sort()  
print(f"sorting time of {size}mb: ", time.time()-start)

**file\_manager.py**

import random  
import os  
import time  
  
  
class File:  
 def \_\_init\_\_(self, path):  
 self.path = path  
 self.file = open(path, "rb")  
 self.prev = (0).to\_bytes(32, byteorder="big")  
 self.curr = self.file.read(32)  
 self.next = self.file.read(32)  
  
 def \_\_next\_\_(self):  
 self.prev = self.curr  
 self.curr = self.next  
 self.next = self.file.read(32)  
  
  
class Writer:  
 def \_\_init\_\_(self, path):  
 self.path = path  
 self.file = open(path, "ab")  
  
  
class Sort:  
 def \_\_init\_\_(self, path, size\_mb, files\_amount):  
 self.path = path  
 self.size = int(size\_mb \* (1024 \*\* 2) / 32)  
 self.files\_amount = files\_amount  
  
 def create\_file(self):  
 print("creating random file...")  
 with open(self.path, "wb") as file:  
 for i in range(self.size):  
 file.write(random.randint(1, 10\_000).to\_bytes(32, byteorder="big"))  
  
 @staticmethod  
 def all\_files\_read(files):  
 return sum([int.from\_bytes(file.curr, byteorder="big") for file in files]) == 0  
  
 @staticmethod  
 def clear\_file(path):  
 open(path, 'w').close()  
  
 @staticmethod  
 def get\_file\_list(path):  
 file = open(path, "rb")  
 arr = []  
 elem = int.from\_bytes(file.read(32), byteorder="big")  
 while elem != 0:  
 arr.append(elem)  
 elem = int.from\_bytes(file.read(32), byteorder="big")  
 return arr  
  
 @staticmethod  
 def get\_file\_size(path):  
 return os.stat(path).st\_size  
  
 def find\_series(self):  
 print("\nfinding series...")  
 start = time.time()  
  
 files = []  
 for i in range(self.files\_amount):  
 x = Writer(f"B{i + 1}.bin")  
 files.append(x)  
  
 a = File(self.path)  
 series = [a.curr]  
 file\_number = 0  
 for i in range(self.size):  
 if a.next >= a.curr:  
 series.append(a.next)  
 else:  
 for elem in series:  
 files[file\_number].file.write(elem)  
  
 file\_number += 1  
 if file\_number == self.files\_amount:  
 file\_number = 0  
  
 series = [a.next]  
 next(a)  
 print("finding series time: ", time.time() - start)  
  
 def merge(self, file\_name1, file\_name2):  
 print("\nmerging...")  
 start = time.time()  
  
 merge\_files = []  
 for i in range(self.files\_amount):  
 self.clear\_file(f"{file\_name2}{i + 1}.bin")  
 x = Writer(f"{file\_name2}{i + 1}.bin")  
 merge\_files.append(x)  
  
 distribute\_files = []  
 for i in range(self.files\_amount):  
 file = File(f"{file\_name1}{i + 1}.bin")  
 distribute\_files.append(file)  
  
 files\_read = False  
 while not files\_read:  
 for m\_file in merge\_files:  
 if self.all\_files\_read(distribute\_files):  
 files\_read = True  
 break  
  
 for file in distribute\_files:  
 file.prev = (0).to\_bytes(32, byteorder="big")  
  
 while True:  
 min\_elem\_file = -1  
 min\_elem = (10001).to\_bytes(32, byteorder="big")  
  
 for i in range(len(distribute\_files)):  
 if distribute\_files[i].prev <= distribute\_files[i].curr <= min\_elem:  
 min\_elem = distribute\_files[i].curr  
 min\_elem\_file = i  
  
 if min\_elem == (10001).to\_bytes(32, byteorder="big"):  
 break  
  
 m\_file.file.write(min\_elem)  
 next(distribute\_files[min\_elem\_file])  
 print("merging time: ", time.time() - start)  
  
 def sort(self):  
 for i in range(self.files\_amount):  
 self.clear\_file(f"B{i + 1}.bin")  
 self.clear\_file(f"C{i + 1}.bin")  
 files = ["B", "C"]  
 self.find\_series()  
 while self.get\_file\_size("file.bin") != self.get\_file\_size("B1.bin") and self.get\_file\_size("file.bin") != self.get\_file\_size("C1.bin"):  
 self.merge(files[0], files[1])  
 files[0], files[1] = files[1], files[0]

## Тестування алгоритму

Задамо розмір файла 10mb, та кількість файлів n = 16. По завершенню роботи алгоритму виведемо деяку кількість перших та останніх чисел із кінцевого файлу, щоб пересвідчитись у тому, що він дійсно відсортований.

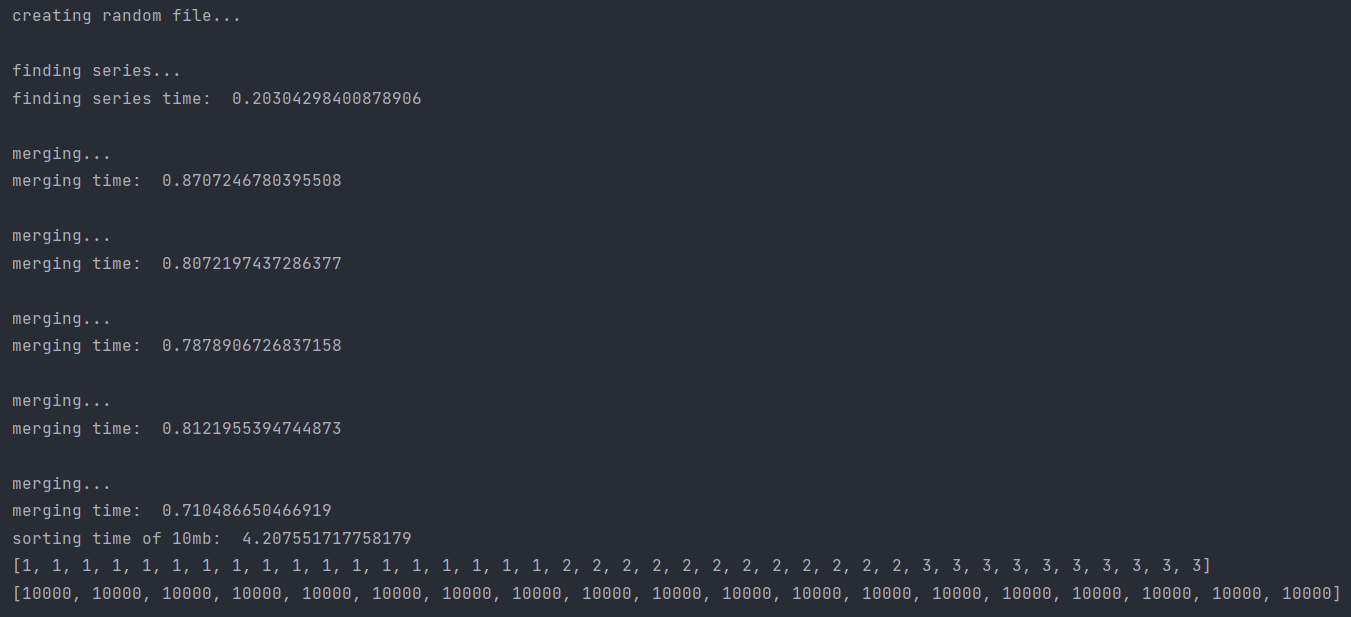


Рис. 3.3 – результат роботи алгоритму

Як можна побачити, алгоритм дійсно відсортував файл розміром 10mb, на що у нього пішло 4.20 секунд.

4. МОДИФІКАЦІЯ АЛГОРИТМУ

4.1 Пояснення

Під час модифікування алгоритму та пришвидшення часу його роботи, були застосовані наступні модифікації:

1. Початковий файл ділиться на k чанків рівного розміру. Кожен із цих чанків поступово зчитується у оперативну пам’ять, та сортується за допомогою

вбудованого у мову програмування Python Timsort, що викликається функцією array.sort(). Кожен відсортований чанк послідовно записуються у бінарні файли B1...Bn.

1. Кількість чанків k, до яких буде застосоване внутрішнє сортування, повинна бути кратна кількості файлів B, та не повинна перевищувати кількість файлів B, піднесену до квадрату (k <= n2). Дана модифікація дозволяє нам проводити процедуру merge лише два рази, де перший – це злиття файлів B1...Bn у файли C1...Ck/n, а другий раз – це злиття файлів C1...Ck/n, у кожному з яких міститься лише одна серія, у файл B1, на чому процедура сортування буде завершена.

4.2 Програмна реалізація

4.2.1 Вихідний код

**main.py**

import time  
from file\_manager import Sort, AdvancedSort  
  
size = 1024  
file = AdvancedSort("file.bin", size, 16, 16\*8)  
file.create\_file()  
start = time.time()  
file.advanced\_sort()  
print(f"sorting time of {size}mb: ", time.time()-start)

**file\_manager.py**

class AdvancedSort(Sort):  
 def \_\_init\_\_(self, path, size\_mb, files\_amount, chunks\_amount):  
 super().\_\_init\_\_(path, size\_mb, files\_amount)  
 self.chunk\_amount = chunks\_amount  
  
 def advanced\_distribution(self):  
 print("\ndistributing...")  
 start = time.time()  
 a = File(self.path)  
 files = []  
 for i in range(self.files\_amount):  
 self.clear\_file(f"B{i + 1}.bin")  
 x = Writer(f"B{i + 1}.bin")  
 files.append(x)  
  
 for i in range(self.chunk\_amount):  
 series = []  
 while a.curr != 0 and len(series) < self.size/self.chunk\_amount:  
 series.append(a.curr)  
 next(a)  
 series.sort()  
 for elem in series:  
 files[i % self.files\_amount].file.write(elem)  
 print("distributing time: ", time.time() - start)  
  
 def advanced\_merge1(self):  
 print("\nmerging...")  
 start = time.time()  
  
 merge\_files = []  
 for i in range(int(self.chunk\_amount/self.files\_amount)):  
 self.clear\_file(f"C{i + 1}.bin")  
 x = Writer(f"C{i + 1}.bin")  
 merge\_files.append(x)  
  
 distribute\_files = []  
 for i in range(self.files\_amount):  
 file = File(f"B{i + 1}.bin")  
 distribute\_files.append(file)  
  
 for j in range(int(self.chunk\_amount/self.files\_amount)):  
  
 for file in distribute\_files:  
 file.prev = (0).to\_bytes(32, byteorder="big")  
  
 while True:  
 min\_elem\_file = -1  
 min\_elem = (10001).to\_bytes(32, byteorder="big")  
  
 for i in range(len(distribute\_files)):  
 if distribute\_files[i].prev <= distribute\_files[i].curr <= min\_elem:  
 min\_elem = distribute\_files[i].curr  
 min\_elem\_file = i  
  
 if min\_elem == (10001).to\_bytes(32, byteorder="big"):  
 break  
  
 merge\_files[j].file.write(min\_elem)  
 next(distribute\_files[min\_elem\_file])  
 print("merging time: ", time.time()-start)  
  
 def advanced\_merge2(self):  
 print("\nmerging...")  
 start = time.time()  
 for i in range(self.files\_amount):  
 self.clear\_file(f"B{i+1}.bin")  
 b\_file = Writer("B1.bin")  
  
 distribute\_files = []  
 for i in range(int(self.chunk\_amount/self.files\_amount)):  
 file = File(f"C{i + 1}.bin")  
 distribute\_files.append(file)  
  
 while True:  
 min\_elem\_file = -1  
 min\_elem = (10001).to\_bytes(32, byteorder="big")  
  
 for i in range(len(distribute\_files)):  
 if distribute\_files[i].prev <= distribute\_files[i].curr <= min\_elem:  
 min\_elem = distribute\_files[i].curr  
 min\_elem\_file = i  
  
 if min\_elem == (10001).to\_bytes(32, byteorder="big"):  
 break  
  
 b\_file.file.write(min\_elem)  
 next(distribute\_files[min\_elem\_file])  
 print("merging time: ", time.time() - start)  
  
 def advanced\_sort(self):  
 for i in range(self.files\_amount):  
 self.clear\_file(f"B{i + 1}.bin")  
 self.clear\_file(f"C{i + 1}.bin")  
 self.advanced\_distribution()  
 self.advanced\_merge1()  
 self.advanced\_merge2()

4.3 Тестування алгоритму

Згенеруємо файл розміром 1024мб = 1гб. Оберемо кількість файлів n = 16, а кількість чанків, які ми будемо сортувати внутрішнім сортуванням– 16\*8 = 128. По завершенню роботи алгоритму виведемо деяку кількість перших і останніх чисел із файлу B1, щоб переконатись, що він дійсно відсортований.

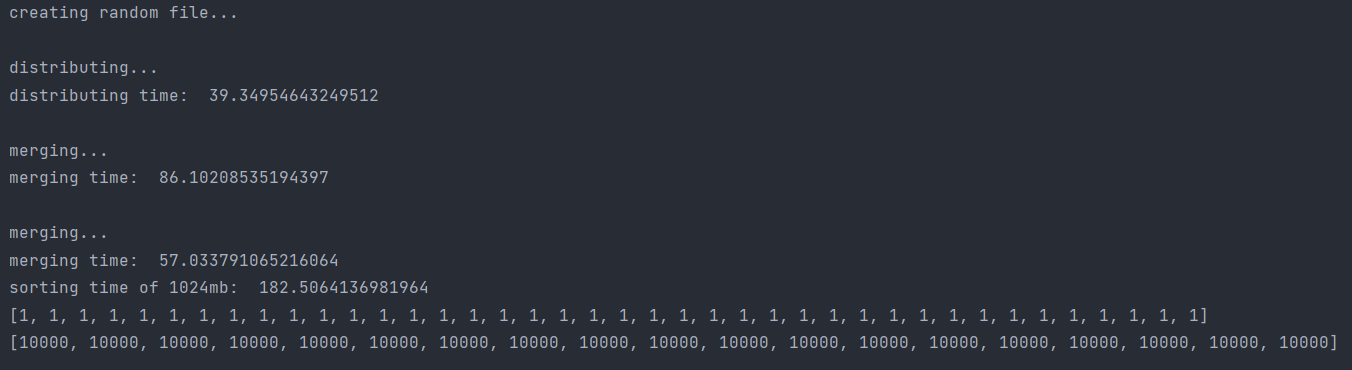


Рис. 4.3.1 – результат роботи програми

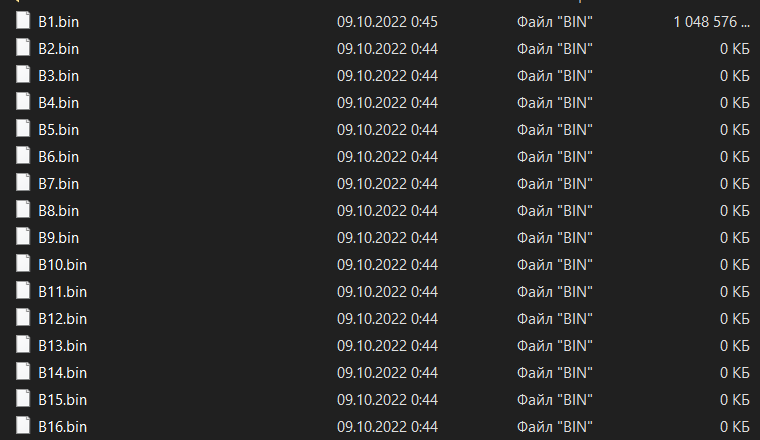


Рис. 4.3.2 – розміри файлів B1-B16

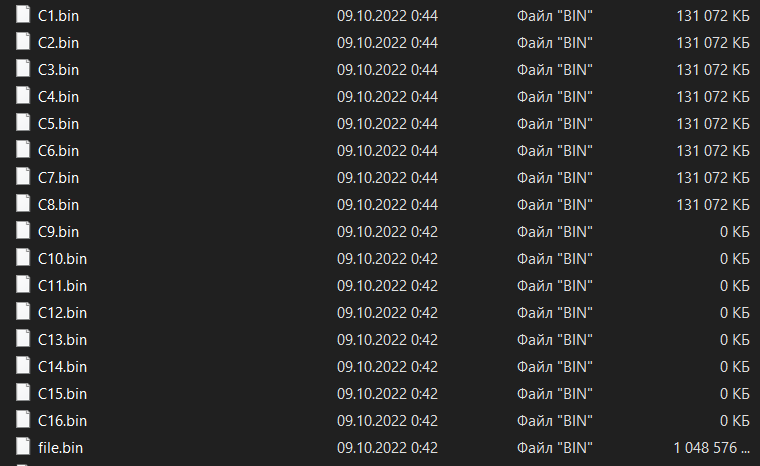


Рис. 4.3.2 – розміри файлів C1-C16 та початкового файлу file.bin

За результатами роботи програми можна побачити, що алгоритм дійсно відсортовує початковий файл розміром 1гб, при чому робить це за час 182.5 секунди = 3.04хв. Алгоритм виконує рівно дві процедури злиття (merge), де перша зливає файли B1...Bn у файли C1...Ck/n, де n – кількість файлів, а k – кількість чанків; а друга процедура зливає файли C1...Ck/n у файл B1. Поглянувши на рисунки 4.3.1 та 4.3.2 можна побачити, що дійсно, файлів С, у яких знаходяться впорядковані серії, рівно 8, тобто k/n = 128/16 = 8. Усі ці файли злились в один файл B1, який займає стільки ж місця, як і початковий файл.

Висновок

При виконанні даної лабораторної роботи було вивчено основні алгоритми зовнішнього сортування та способи їх модифікації, реалізовано алгоритм багатошляхового збалансованого сортування.

Даний алгоритм зовнішнього сортування умовно можна розділити на два етапи: перший – розбиття початкового файлу на серії і розподіл серій по файлам; другий – злиття серій із одних файлів у інші. Виконані модифікації пришвидшують швидкість роботи обох із цих етапів.

1. Зчитування, сортування і запис серій фіксованого розміру і кількості. Дана модифікація виявилась найбільш ефективною, і дозволила пришвидшити час виконання алгоритму багатошляхового збалансованого сортування приблизно у 3 рази. Завдяки цій модифікації було значно зменшено кількість виконання процедури злиття (merge), яка займала абсолютну більшість часу виконання програми.
2. Формування кількості чанків, що кратна кількості файлів і не перевершує її квадрату (k <= n2). Дана модифікація дозволяє зменшити кількість процедур злиття до двох, і при цьому уникнути значної кількості перевірок, які потрібні при немодифікованому алгоритмі. Дана модифікація пришвидшила час виконання алгоритму ще на 15-20%.

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%.