**Міністерство освіти і науки України**

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

**Факультет інформатики та обчислювальної техніки**

**Кафедра ІПІ**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Алгоритми та структури даних 2. Структури даних»

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

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-11 Трикош Іван Володимирович*

**Перевірив(ла)**

(прізвище, ім'я, по батькові)

*Халус Олена Андріївна*

Київ 2022

# Мета лабораторної роботи

Мета роботи – вивчити основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування і оцінити поріг їх ефективності.

# ЗаВдання

Виконати аналіз алгоритму внутрішнього сортування на відповідність наступним властивостям (таблиця 2.1):

* стійкість;
* «природність» поведінки (Adaptability);
* базуються на порівняннях;
* необхідність додаткової пам'яті (об'єму);
* необхідність в знаннях про структуру даних.

Записати алгоритм внутрішнього сортування за допомогою псевдокоду (чи іншого способу по вибору).

Провести аналіз часової складності в гіршому, кращому і середньому випадках та записати часову складність в асимптотичних оцінках.

Виконати програмну реалізацію алгоритму на будь-якій мові програмування з фіксацією часових характеристик оцінювання (кількість порівнянь, кількість перестановок, глибина рекурсивного поглиблення та інше в залежності від алгоритму).

Провести ряд випробувань алгоритму на масивах різної розмірності (10, 100, 1000, 5000, 10000, 20000, 50000 елементів) і різних наборів вхідних даних (впорядкований масив, зворотно упорядкований масив, масив випадкових чисел) і побудувати графіки залежності часових характеристик оцінювання від розмірності масиву, нанести на графік асимптотичну оцінку гіршого і кращого випадків для порівняння.

Зробити порівняльний аналіз двох алгоритмів.

Зробити узагальнений висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм сортування** |
| 1 | Сортування бульбашкою |
| 2 | Сортування гребінцем («розчіскою») |

# Виконання

*СОРТУВАННЯ БУЛЬБАШКОЮ*

## Аналіз алгоритму на відповідність властивостям

Аналіз алгоритму сортування бульбашкою на відповідність властивостям наведено в таблиці 3.1.

Таблиця 3.1 – Аналіз алгоритму на відповідність властивостям

|  |  |
| --- | --- |
| **Властивість** | **Сортування бульбашкою** |
| Стійкість | Стійкий |
| «Природність» поведінки (Adaptability) | Не має природної поведінки |
| Базуються на порівняннях | Так |
| Необхідність в додатковій пам'яті (об'єм) | O(1) |
| Необхідність в знаннях про структури даних | Потрібні елементарні знання |

## Псевдокод алгоритму

і = 1

**повторити** n **разів**

j = 0

**повторити** n - 1 **разів**

**якщо** array[j] > array[j+1] **то**

buffer = array[j+1]

array[j+1] = array[j]

array[j] = buffer

**все якщо**

j = j + 1

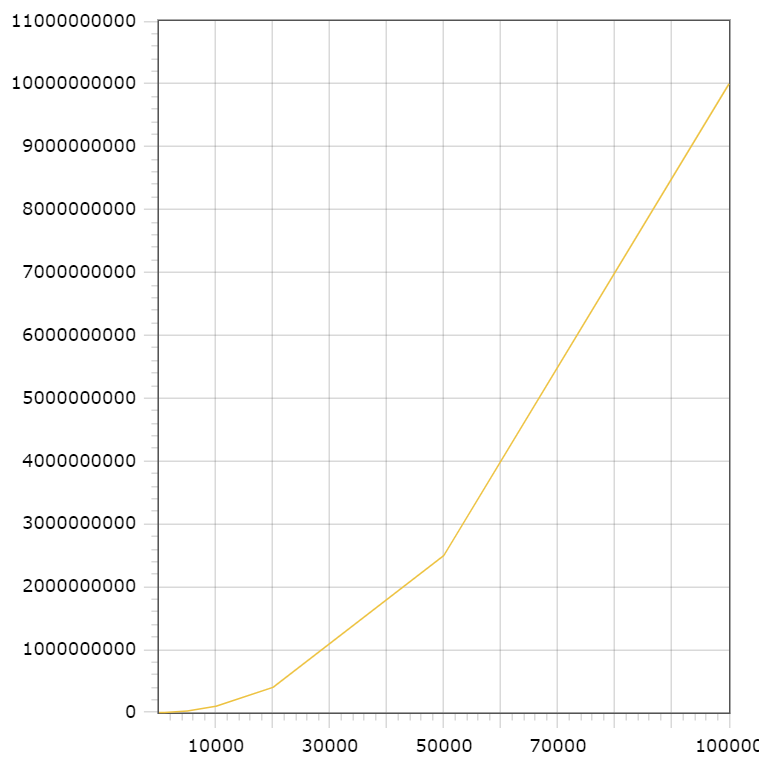
**все повторити**

i = i + 1

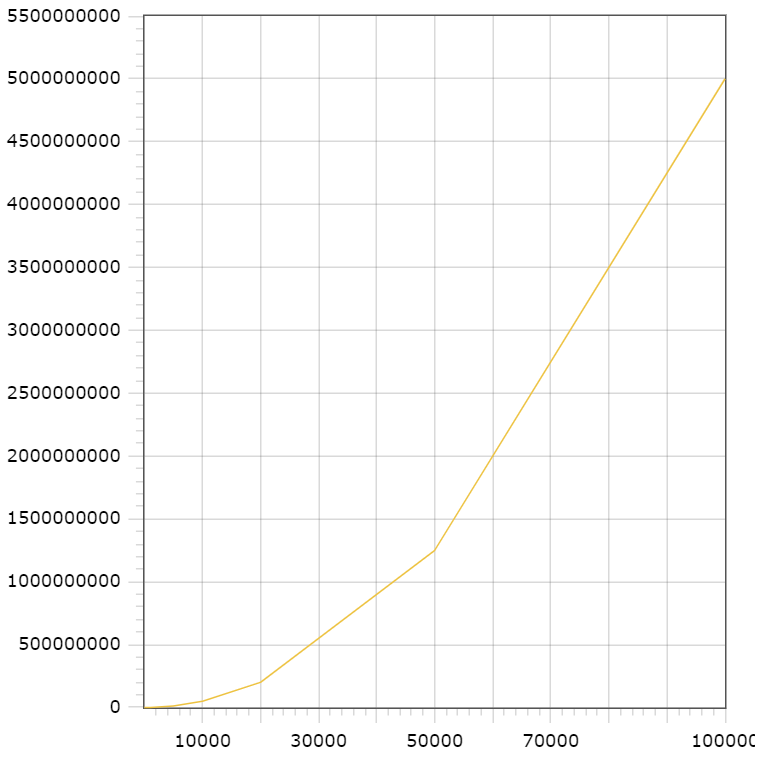
**все повторити**

## Аналіз часової складності

## Графік залежності кількісті порівнянь від розміру масиву:



Графік залежності кількості перестановок від розміру масиву:



Бачимо, що загальна кількість операцій, якщо знехтувати константами, дорівнює n2.

## Програмна реалізація алгоритму

### Вихідний код

def buble\_sort(array):

size = len(array)

counter1 = 0 #Для порівнянь

counter2 = 0 #Для перестановок

for i in range(size):

for j in range(0, size - 1):

counter1 += 1

if array[j] > array[j+1]:

#Міняємо елементи місцями, якщо попередній елемент більший за поточний

counter2 += 1

array[j], array[j+1] = array[j+1] , array[j]

return array, counter1, counter2

### Приклад роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми сортування масивів на 100 і 1000 елементів відповідно.

Рисунок 3.1 – Сортування масиву на 100 елементів

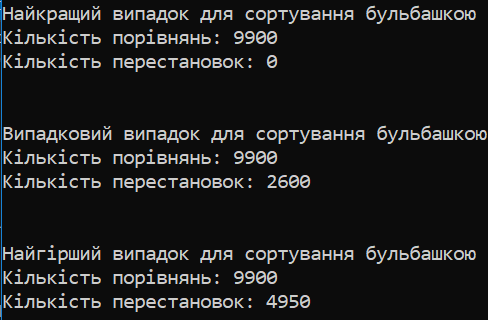
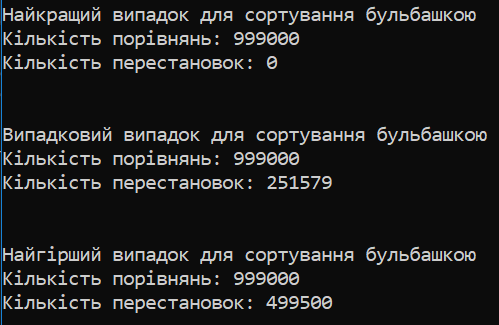


Рисунок 3.2 – Сортування масиву на 1000 елементів



## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.2 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування бульбашки для масивів різної розмірності, коли масив містить упорядковану послідовність елементів.

Таблиця 3.2 – Характеристики оцінювання алгоритму сортування бульбашки для упорядкованої послідовності елементів у масиві

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 90 | 0 |
| 100 | 9900 | 0 |
| 1000 | 999000 | 0 |
| 5000 | 24995000 | 0 |
| 10000 | 99990000 | 0 |
| 20000 | 399980000 | 0 |
| 50000 | 2499950000 | 0 |

В таблиці 3.3 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування бульбашки для масивів різної розмірності, коли масиви містять зворотно упорядковану послідовність елементів.

Таблиця 3.3 – Характеристики оцінювання алгоритму сортування бульбашки для зворотно упорядкованої послідовності елементів у масиві.

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 90 | 45 |
| 100 | 9900 | 4950 |
| 1000 | 999000 | 499500 |
| 5000 | 24995000 | 12497500 |
| 10000 | 99990000 | 49995000 |
| 20000 | 399980000 | 199990000 |
| 50000 | 2499950000 | 1249975000 |

У таблиці 3.4 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування бульбашки для масивів різної розмірності, масиви містять випадкову послідовність елементів.

Таблиця 3.4 – Характеристика оцінювання алгоритму сортування бульбашки для випадкової послідовності елементів у масиві.

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 90 | 19 |
| 100 | 9900 | 2311 |
| 1000 | 999000 | 256714 |
| 5000 | 24995000 | 6204176 |
| 10000 | 99990000 | 24674089 |
| 20000 | 399980000 | 100415512 |
| 50000 | 2499950000 | 628060921 |

### Графіки залежності часових характеристик оцінювання від розмірності масиву

На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву для випадків, коли масиви містять упорядковану послідовність елементів (зелений графік), коли масиви містять зворотно упорядковану послідовність елементів (червоний графік), коли масиви містять випадкову послідовність елементів (синій графік), також показані асимптотичні оцінки гіршого (фіолетовий графік) і кращого (жовтий графік) випадків для порівняння.

Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

*СОРТУВАННЯ ГРЕБІНЦЕМ*

## Аналіз алгоритму на відповідність властивостям

Аналіз алгоритму сортування гребінцем на відповідність властивостям наведено в таблиці 3.1.

Таблиця 3.1 – Аналіз алгоритму на відповідність властивостям

|  |  |
| --- | --- |
| **Властивість** | **Сортування бульбашкою** |
| Стійкість | Нестійкий |
| «Природність» поведінки (Adaptability) | Не має природної поведінки |
| Базуються на порівняннях | Так |
| Необхідність в додатковій пам'яті (об'єм) | O(1) |
| Необхідність в знаннях про структури даних | Потрібні елементарні знання |

## Псевдокод алгоритму

step = len(array)

**поки** step > 1 **повторити**

step = int(step / 1,247)

**якщо** step < 1 **то**

step = 1

**все якщо**

i = 0

**повторити** len(array) – step **разів**

**якщо** array[і] > array[і+1] **то**

buffer = array[і+1]

array[і+1] = array[і]

array[і] = buffer

**все якщо**

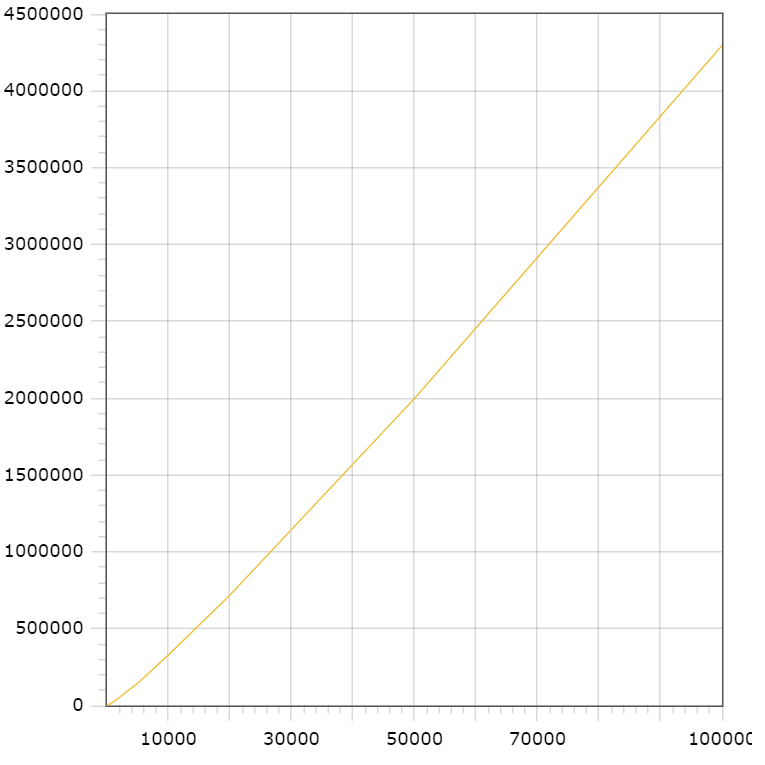
i += 1

**все повторити**

**все повторити**

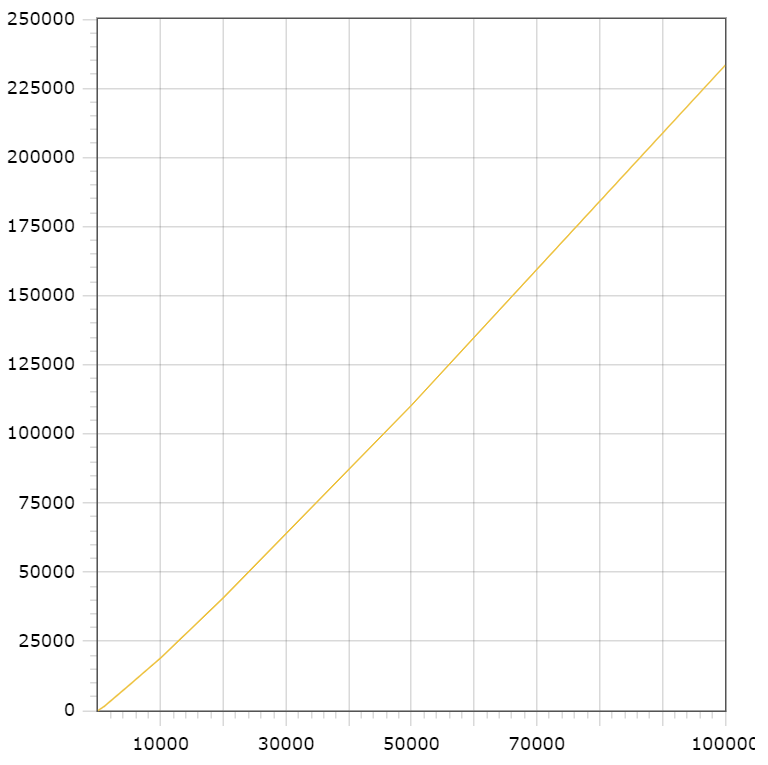
## Аналіз часової складності

Побудуємо графік залежності кількісті порівнянь від розміру масиву у найгіршому випадку.



З графіку бачимо, що час виконання порівнянь приблизно дорівнює c1\*n\*log(n), де c1 – час виконання одного порівняння.

Побудуємо графік залежності кількісті перестановок від розміру масиву у найгіршому випадку.



З графіку бачимо, що час виконання перестановок приблизно дорівнює c2\*n\*log(n), де c2 – час виконання однієї перестановки. Іншими операціями циклу, як і константами, можна знехтувати. Отже, часова складність алгоритму дорівнює n\*log(n).

## Програмна реалізація алгоритму

### Вихідний код

def comb\_sort(array):

size = len(array)

step = len(array)

counter1 = 0 #Для порівнянь

counter2 = 0 #Для перестановок

while step > 1:

step = int(step / 1.247)

if step < 1:

step = 1

for j in range(0, size - step):

counter1 += 1

if array[j] > array[j+step]:

#Міняємо елементи місцями, якщо попередній елемент більший за поточний

counter2 += 1

array[j], array[j+step] = array[j+step] , array[j]

return array, counter1, counter2

### Приклад роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми сортування масивів на 100 і 1000 елементів відповідно.

Рисунок 3.1 – Сортування масиву на 100 елементів

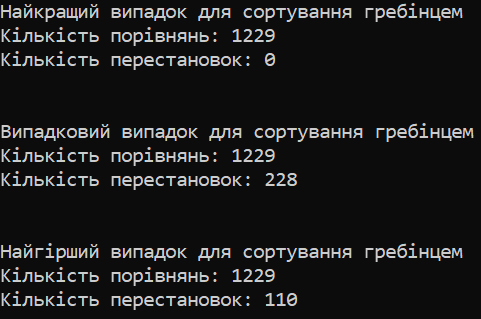
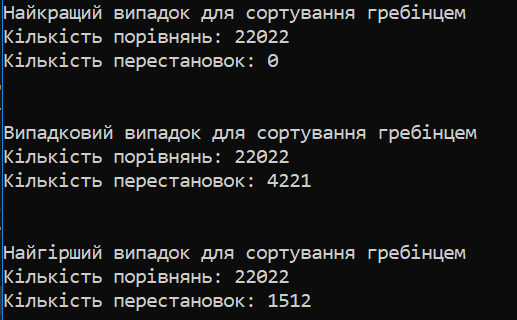


Рисунок 3.2 – Сортування масиву на 1000 елементів



## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.2 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування гребінцем для масивів різної розмірності, коли масив містить упорядковану послідовність елементів.

Таблиця 3.2 – Характеристики оцінювання алгоритму сортування гребінцем для упорядкованої послідовності елементів у масиві

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 36 | 0 |
| 100 | 1229 | 0 |
| 1000 | 22022 | 0 |
| 5000 | 144832 | 0 |
| 10000 | 329598 | 0 |
| 20000 | 719136 | 0 |
| 50000 | 1997680 | 0 |

В таблиці 3.3 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування гребінцем для масивів різної розмірності, коли масиви містять зворотно упорядковану послідовність елементів.

Таблиця 3.3 – Характеристики оцінювання алгоритму сортування гребінцем для зворотно упорядкованої послідовності елементів у масиві.

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 36 | 9 |
| 100 | 1229 | 110 |
| 1000 | 22022 | 1512 |
| 5000 | 144832 | 9154 |
| 10000 | 329598 | 19018 |
| 20000 | 719136 | 40730 |
| 50000 | 1997680 | 110332 |

У таблиці 3.4 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування гребінцем для масивів різної розмірності, масиви містять випадкову послідовність елементів.

Таблиця 3.4 – Характеристика оцінювання алгоритму сортування гребінцем для випадкової послідовності елементів у масиві.

|  |  |  |
| --- | --- | --- |
| Розмірність масиву | Число порівнянь | Число перестановок |
| 10 | 36 | 9 |
| 100 | 1229 | 238 |
| 1000 | 22022 | 4403 |
| 5000 | 144832 | 27310 |
| 10000 | 329598 | 59652 |
| 20000 | 719136 | 130649 |
| 50000 | 1997680 | 367940 |

### Графіки залежності часових характеристик оцінювання від розмірності масиву

На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву для випадків, коли масиви містять упорядковану послідовність елементів (зелений графік), коли масиви містять зворотно упорядковану послідовність елементів (червоний графік), коли масиви містять випадкову послідовність елементів (синій графік), також показані асимптотичні оцінки гіршого (фіолетовий графік) і кращого (жовтий графік) випадків для порівняння.

Рисунок 3.3 – графіки залежності часових характеристик оцінювання

ПРОГРАМНИЙ КОД

import random

def buble\_sort(array):

size = len(array)

counter1 = 0 #Для порівнянь

counter2 = 0 #Для перестановок

for i in range(size):

for j in range(0, size - 1):

counter1 += 1

if array[j] > array[j+1]:

#Міняємо елементи місцями, якщо попередній елемент більший за поточний

counter2 += 1

array[j], array[j+1] = array[j+1] , array[j]

return array, counter1, counter2

def comb\_sort(array):

size = len(array)

step = len(array)

counter1 = 0 #Для порівнянь

counter2 = 0 #Для перестановок

while step > 1:

step = int(step / 1.247)

if step < 1:

step = 1

for j in range(0, size - step):

counter1 += 1

if array[j] > array[j+step]:

#Міняємо елементи місцями, якщо попередній елемент більший за поточний

counter2 += 1

array[j], array[j+step] = array[j+step] , array[j]

return array, counter1, counter2

def test\_buble(n):

bestcase = [i for i in range(1, n + 1)]

bestcase1 = bestcase[:]

bestcase, counter1, counter2 = buble\_sort(bestcase)

output(bestcase1, bestcase, counter1, counter2, n, "Найкращий", "бульбашкою")

randomcase = random.sample(range(1, n + 1), n)

randcase = randomcase[:]

randomcase, counter1, counter2 = buble\_sort(randomcase)

output(randcase, randomcase, counter1, counter2, n, "Випадковий", "бульбашкою")

worstcase = [i for i in range(n, 0, -1)]

worstcase1 = worstcase[:]

worstcase, counter1, counter2 = buble\_sort(worstcase)

output(worstcase1, worstcase, counter1, counter2, n, "Найгірший", "бульбашкою")

print("\n\n")

def test\_comb(n):

bestcase = [i for i in range(1, n + 1)]

bestcase1 = bestcase[:]

bestcase, counter1, counter2 = comb\_sort(bestcase)

output(bestcase1, bestcase, counter1, counter2, n, "Найкращий", "гребінцем")

randomcase = random.sample(range(1, n + 1), n)

randcase = randomcase[:]

randomcase, counter1, counter2 = comb\_sort(randomcase)

output(randcase, randomcase, counter1, counter2, n, "Випадковий", "гребінцем")

worstcase = [i for i in range(n, 0, -1)]

worstcase1 = worstcase[:]

worstcase, counter1, counter2 = comb\_sort(worstcase)

output(worstcase1, worstcase, counter1, counter2, n, "Найгірший", "гребінцем")

print("\n")

def output(case1, case2, counter1, counter2, n, string1, string2):

print()

print("\n", string1," випадок для сортування ", string2, sep = "")

if n <= 10:

print("Вхідний масив\n", case1, "\nВихідний масив\n", case2)

print("Кількість порівнянь:", counter1)

print("Кількість перестановок:", counter2)

def main():

n = 50000

test\_buble(n)

test\_comb(n)

main()

Висновок

При виконанні даної лабораторної роботи я вивчив основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування і оцінив поріг їх ефективності. Я визначив складність алгоритму сортування бульбашкою (n2) та алгоритму сортування гребінцем n\*log(n) і дійшов висновку, що алгоритм сортування гребінцем є значно ефективнішим, ніж алгоритм сортування бульбашкою, довівши це теоретично та практично. Я вивчив основні методи дослідження алгоритмів.

array = [3, 5, 4, 1, 2]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5 | 4 | 1 | 2 |
| 3 | 5 | 4 | 1 | 2 |
| 3 | 4 | 5 | 1 | 2 |
| 3 | 4 | 1 | 5 | 2 |
| 3 | 4 | 1 | 2 | 5 |
| 3 | 4 | 1 | 2 | 5 |
| 3 | 1 | 4 | 2 | 5 |
| 3 | 1 | 2 | 4 | 5 |
| 3 | 1 | 2 | 4 | 5 |
| 1 | 3 | 2 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |