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) *ІП-15. Борисик Владислав Тарасович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив Соколовський Владислав Володимирович _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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»)

3 ВИКОНАННЯ

3.1 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	Так, алгоритм є стійким
«Природність» поведінки (Adaptability)	Так, алгоритм є природним
Базуються на порівняннях	Так, алгоритм базується на порівняннях
Необхідність в додатковій пам'яті (об'єм)	Ні, алгоритм не потребує додаткової пам’яті
Необхідність в знаннях про структури даних	Так, для використання цього алгоритму потрібно мати базові знання про структури даних

3.2 Псевдокод алгоритму

Підпрограма bubble_sort(array):
Початок
array_size := array.size()
is_swtiched := True
i := 0
повторити поки i < array_size - 2 **і** is_swtiched == True
is_switched := False
повторити для j **від** 0 **до** array_size-i-1
якщо array [j] > array [j + 1]
то

is_switched := True

temp := array[j]

array[j] := array[j + 1]

array[j + 1] := temp

все повторити

i += 1

все повторити

Кінець

3.3 Аналіз часової складності

Найгірша швидкодія: $O(n^2)$

Найкраща швидкодія: $O(n)$

Середня швидкодія: $O(n^2)$

3.4 Програмна реалізація алгоритму

```
def bubble_sort(list_to_sort: list[int]):
    list_size = len(list_to_sort)
    is_switched = True
    i = 0

    while is_switched and i < list_size - 2:
        is_switched = False

        for j in range(list_size - i - 1):
            if list_to_sort[j] > list_to_sort[j + 1]:
                is_switched = True
                temp = list_to_sort[j]
                list_to_sort[j] = list_to_sort[j + 1]
                list_to_sort[j + 1] = temp
        i += 1
```

3.4.1 Вихідний код

```
def bubble_sort(list_to_sort: list[int]):
    list_size = len(list_to_sort)
```

```
    is_switched = True
    i = 0

    while is_switched and i < list_size - 2:
        is_switched = False

        for j in range(list_size - i - 1):
            if list_to_sort[j] > list_to_sort[j + 1]:
                is_switched = True
                temp = list_to_sort[j]
                list_to_sort[j] = list_to_sort[j + 1]
                list_to_sort[j + 1] = temp
        i += 1
```

3.4.2 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

```
[37, 21, 71, 39, 53, 60, 40, 89, 58, 33, 19, 10, 84, 69, 95, 11, 72, 3, 15, 57, 59, 20, 50, 93, 35, 51, 54, 63, 27, 52, 46, 4, 48, 65, 28, 45, 38,
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
38, 96, 98, 94, 76, 80, 90, 70, 92, 67, 18, 79, 1, 23, 97, 81, 6, 62, 64, 30, 43, 24, 41, 87, 17, 82, 8, 85, 42, 66, 61, 47, 86, 68, 25, 32, 77, 1
 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 7
85, 42, 66, 61, 47, 86, 68, 25, 32, 77, 14, 91, 7, 12, 2, 55, 75, 22, 88, 36, 99, 29, 49, 5, 74, 13, 44, 31, 9, 34, 78, 26, 100, 56, 16, 83, 73]
65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

Рисунок 3.2 – Сортування масиву на 1000 елементів

```
[861, 442, 548, 782, 964, 505, 935, 51, 437, 86, 997, 243, 321, 687, 481, 873, 680, 469, 946, 132, 280, 978, 88, 202, 595, 182, 42, 201, 379, 961,
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
1, 887, 843, 813, 103, 568, 227, 735, 624, 728, 430, 608, 356, 387, 174, 991, 209, 108, 910, 857, 757, 922, 986, 493, 784, 507, 727, 144, 500, 169
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75
701, 874, 413, 148, 391, 883, 322, 533, 707, 127, 363, 12, 589, 972, 634, 825, 192, 770, 858, 694, 844, 747, 474, 28, 593, 214, 840, 92, 524, 704,
, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972,
04, 609, 649, 207, 350, 574, 679, 64, 425, 697, 929, 380, 38, 269, 306, 422, 79, 191, 718, 582, 181, 517, 729, 236, 711, 924, 725, 597, 395, 81]
72, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000]
```

3.5 Тестування алгоритму

Як було видно у попередньому пункті (п. 3.4.2), алгоритм правильно відсортував випадково згенерований масив на 100 і 1000 елементів, які не повторювались. Додатково протестуємо алгоритм сортування бульбашкою на випадково згенерованому масиві з розмірністю 20 елементів (елементи можуть повторюватись).

Код:

```python
import random


def bubble_sort(list_to_sort: list[int]):
    list_size = len(list_to_sort)
    is_switched = True
    i = 0

    while is_switched and i < list_size - 2:
        is_switched = False

        for j in range(list_size - i - 1):
            if list_to_sort[j] > list_to_sort[j + 1]:
                is_switched = True
                temp = list_to_sort[j]
                list_to_sort[j] = list_to_sort[j + 1]
                list_to_sort[j + 1] = temp
        i += 1


# кількість елементів у масиві
n = 20

random_list = [random.randint(1, n) for i in range(n)]

print(random_list)
bubble_sort(random_list)
print(random_list)
```

Результат тестування:

```
[1, 2, 2, 19, 19, 8, 6, 20, 2, 1, 13, 14, 14, 18, 7, 11, 7, 9, 15, 1]
[1, 1, 1, 2, 2, 2, 6, 7, 7, 8, 9, 11, 13, 14, 14, 15, 18, 19, 19, 20]
```

Як видно на скриншоті, алгоритм коректно відсортував випадково згенерований масив. Отже, наш алгоритм сортування бульбашкою працює правильно.

3.5.1 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	9	0
100	999	0
1000	999	0
5000	4999	0
10000	9999	0
20000	19999	0
50000	49999	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	44	44
100	4 949	4 949
1000	499 499	499 499
5000	12 497 499	12 4974 99
10000	49 994 999	49 994 999
20000	199 989 999	199 989 999
50000	1 249 974 999	1 249 974 999

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

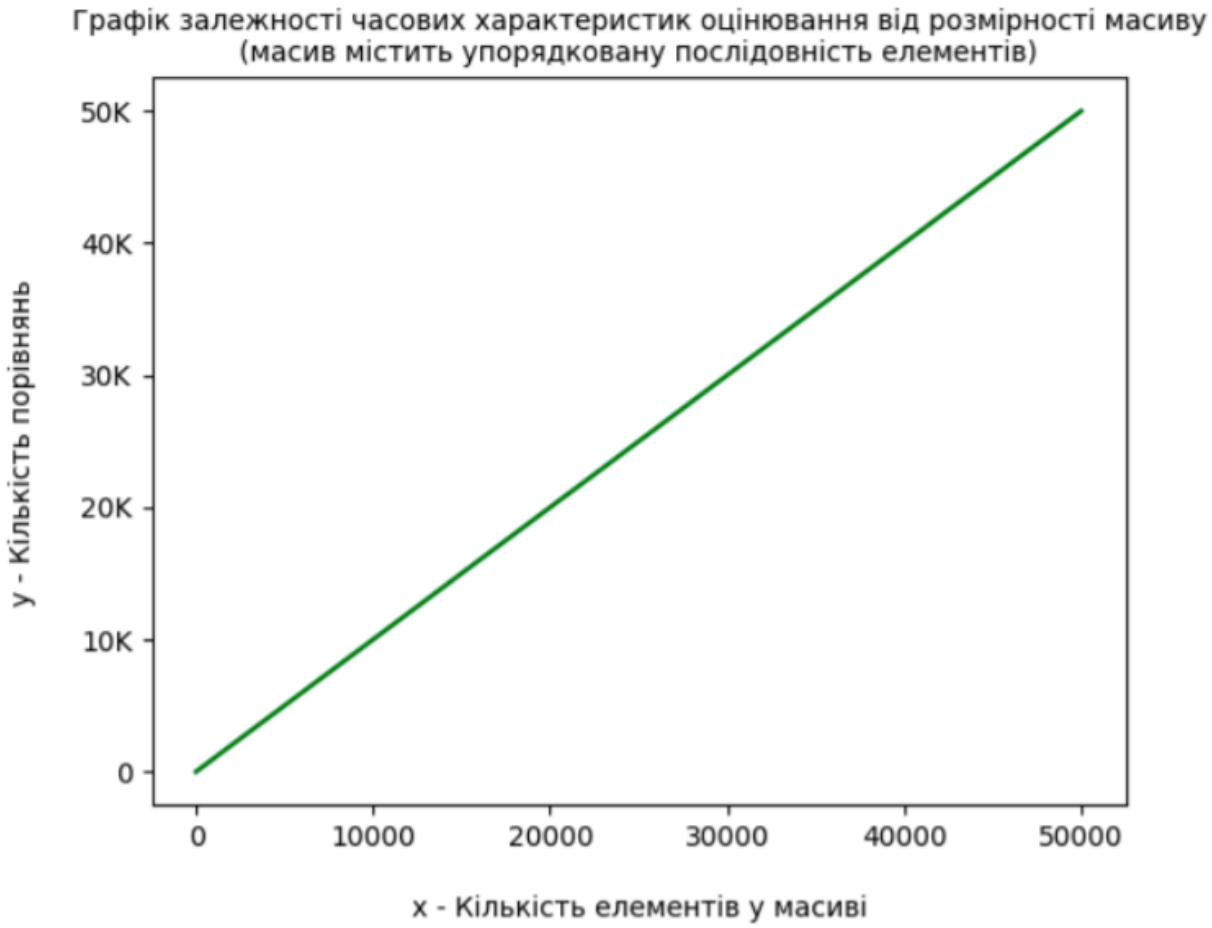
Таблиця 3.4 – Характеристика оцінювання алгоритму сортування бульбашки для випадкової послідовності елементів у масиві.

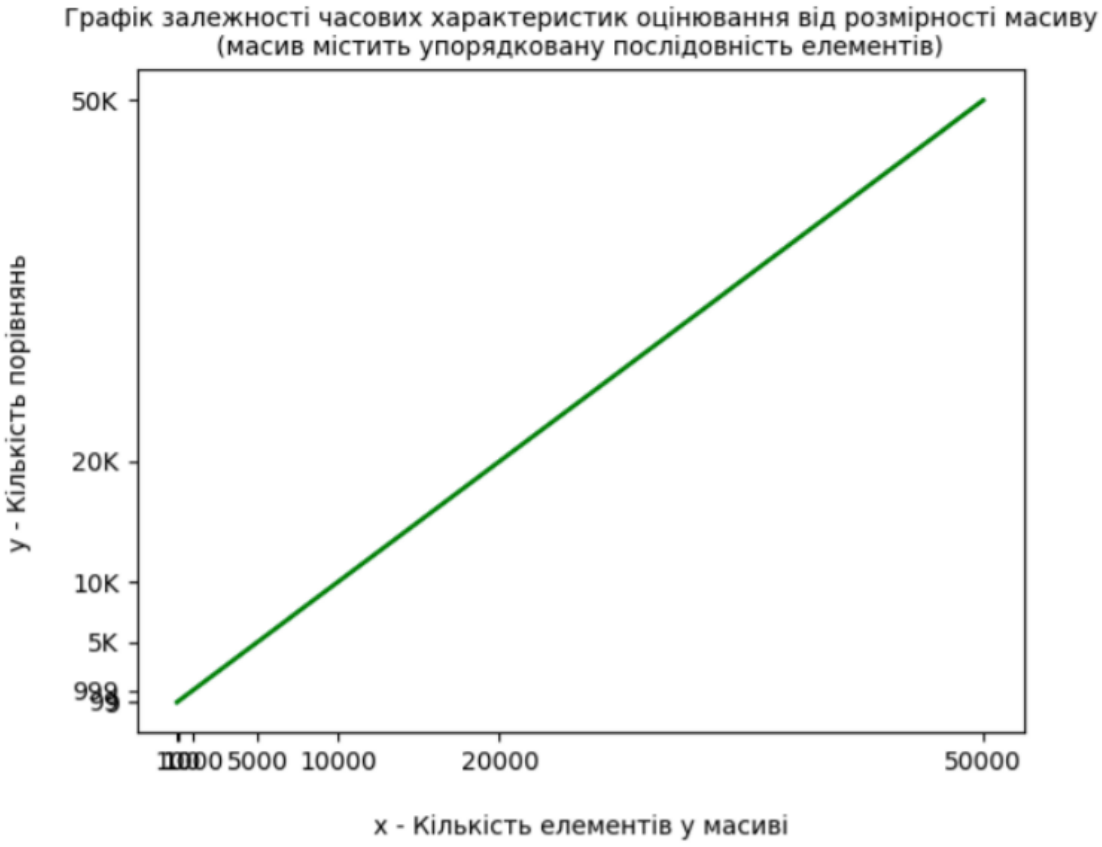
Розмірність масиву	Число порівнянь	Число перестановок
10	42	21
100	4 935	2 520
1000	497 847	248 833
5000	12 497 347	6 267 487
10000	49 953 959	24 777 211
20000	199 989 054	100 893 798
50000	1 249 945 839	626 589 657

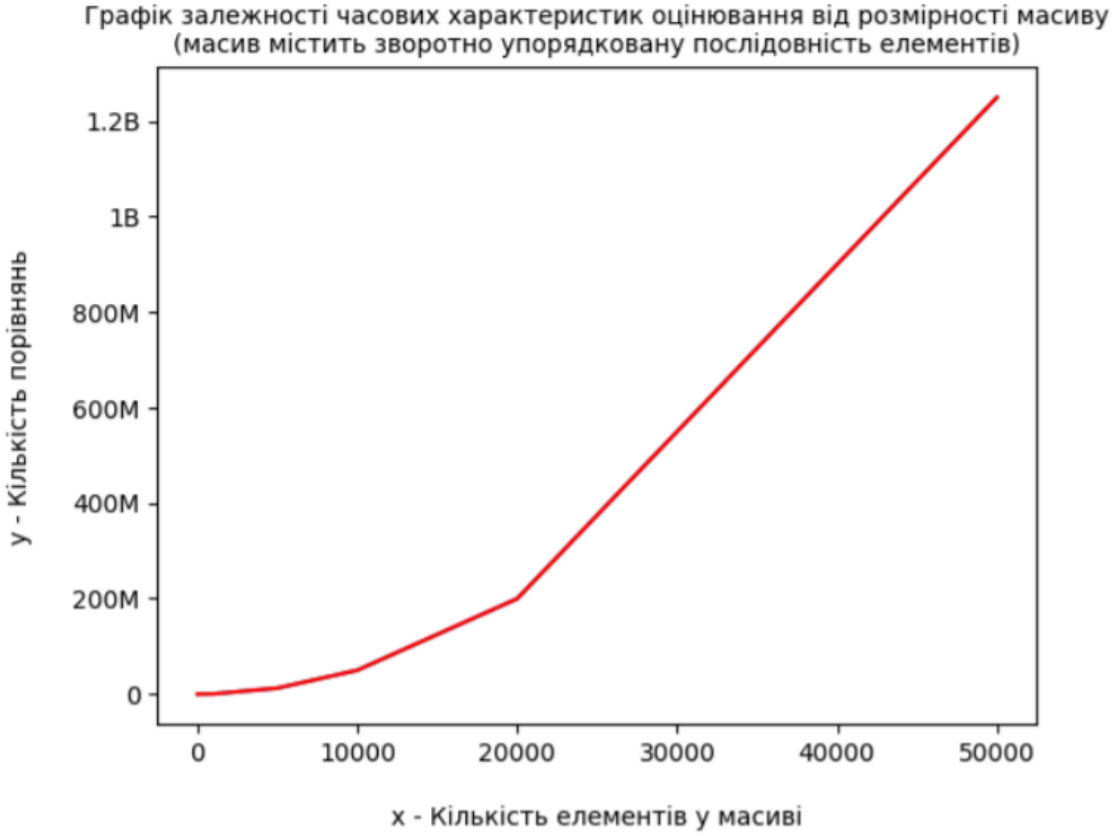
3.5.2 Графіки залежності часових характеристик оцінювання від розмірності масиву

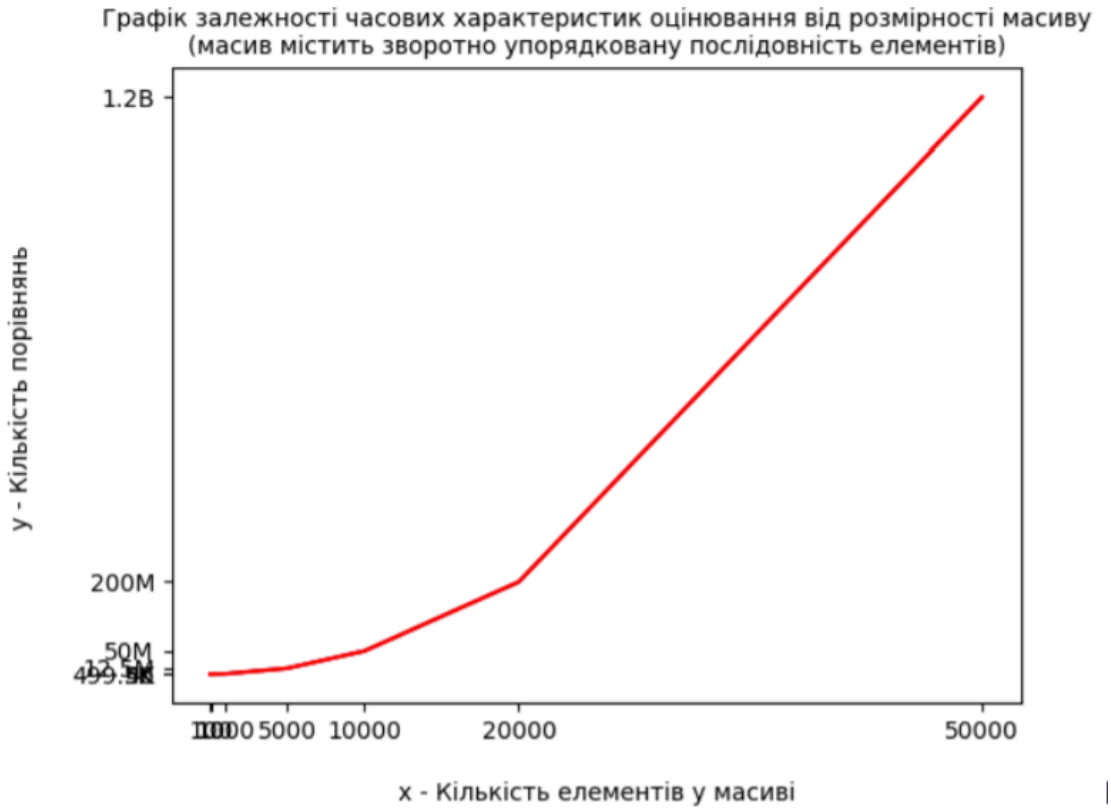
На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву для випадків, коли масиви містять упорядковану послідовність елементів (зелений графік), коли масиви містять зворотно упорядковану послідовність елементів (червоний графік), коли масиви містять випадкову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

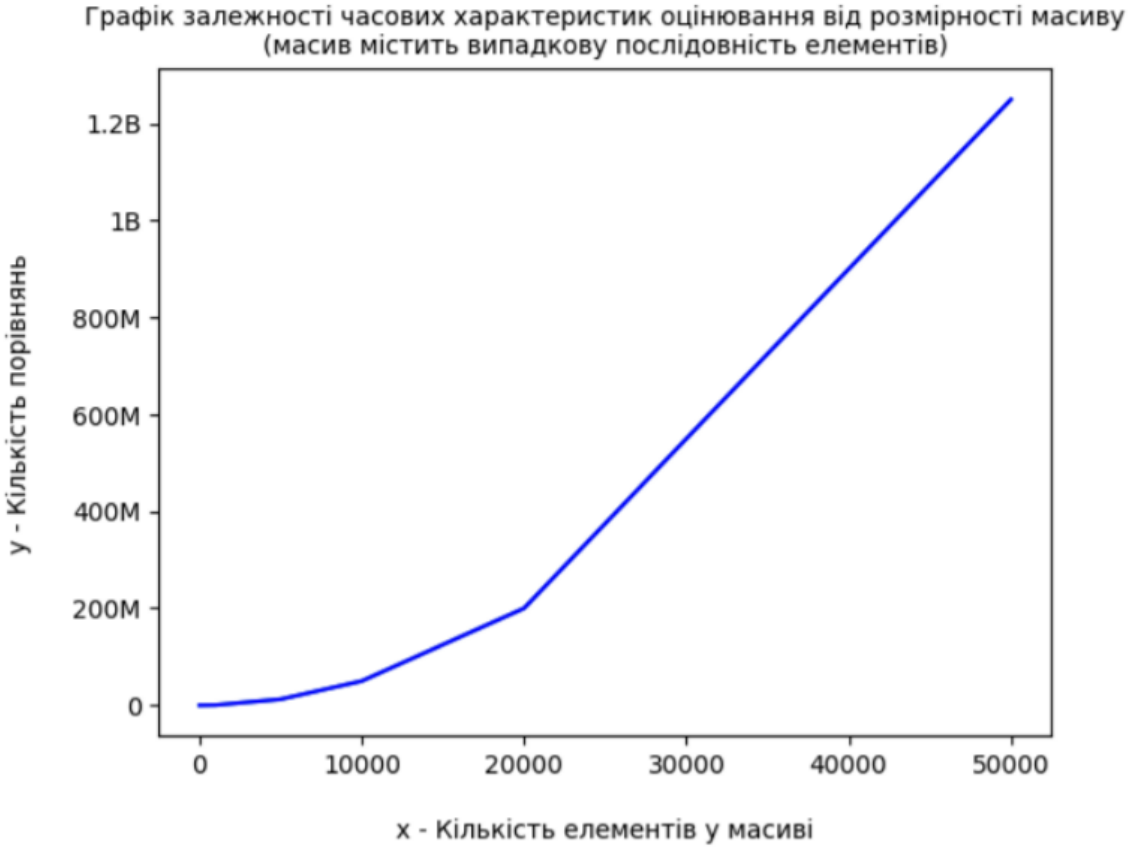
Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

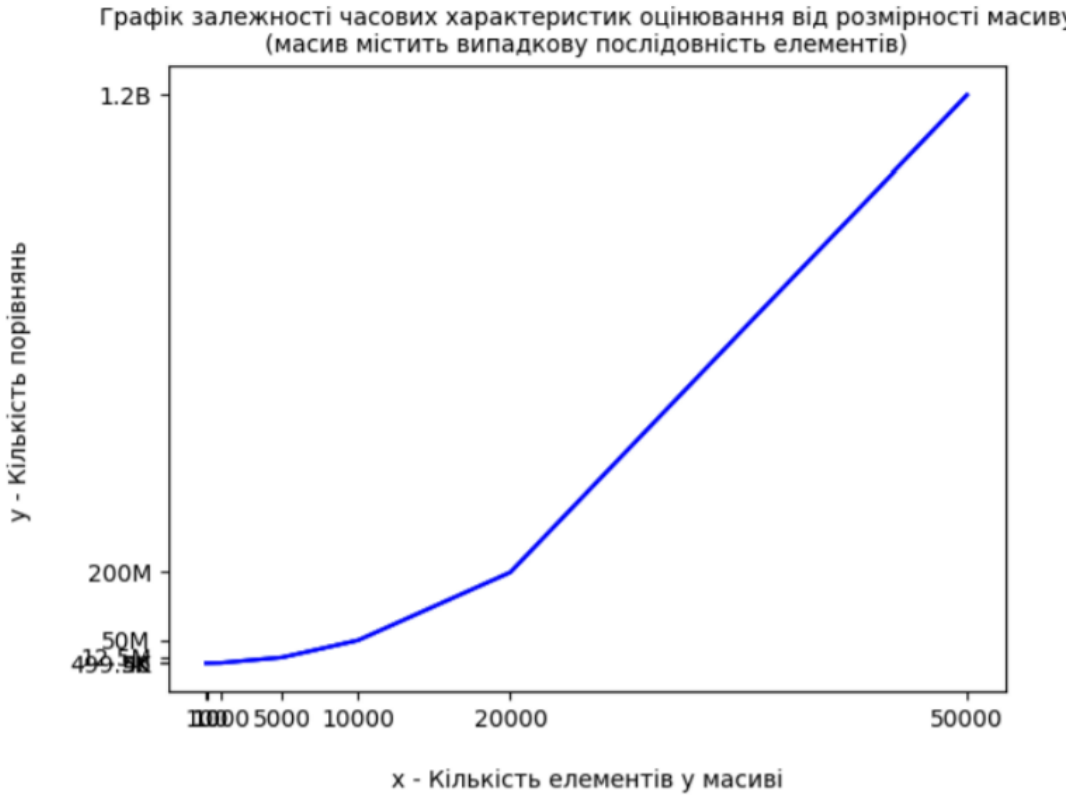



Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить упорядковану послідовність елементів)
y - Кількість порівнянь
50K
20K
10K
5K
5000
10000
20000
50000
x - Кількість елементів у масиві

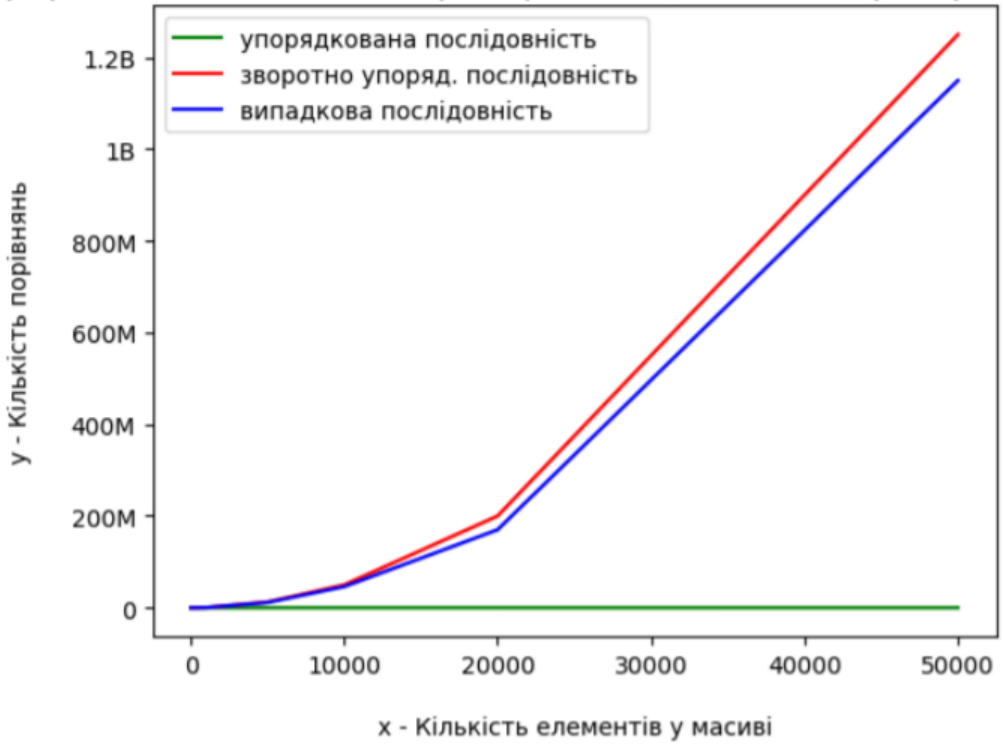

Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить зворотно упорядковану послідовність елементів)
1.2B
1B
800M
600M
400M
200M
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві

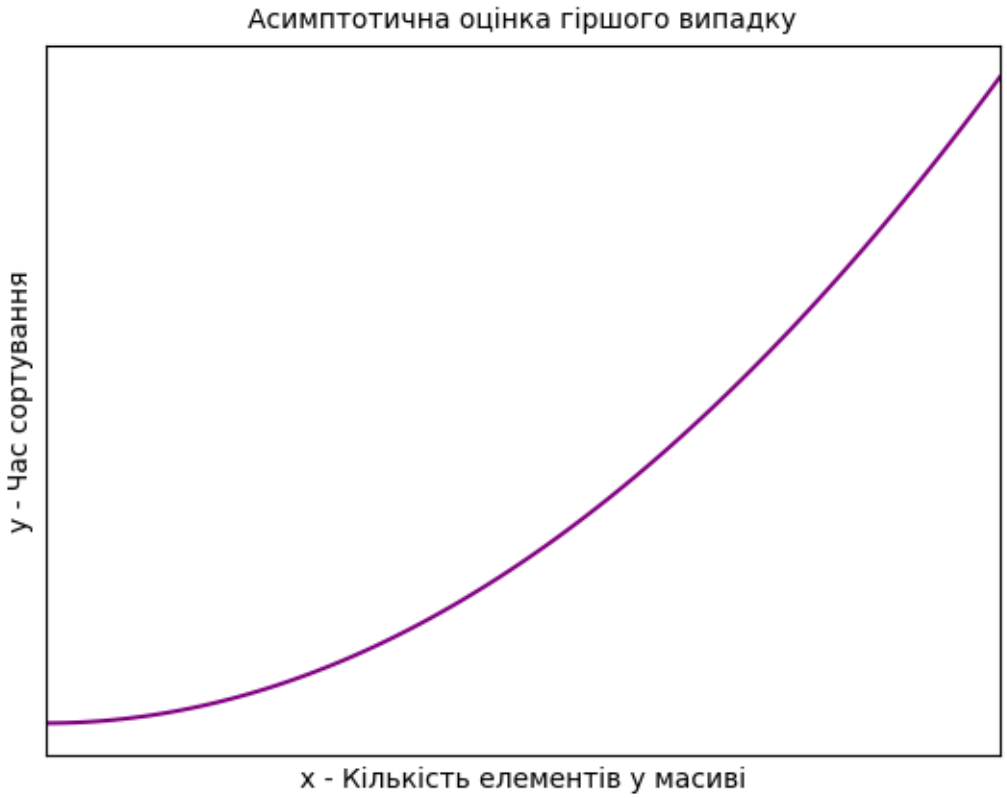

Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить зворотно упорядковану послідовність елементів)
1.2B
200M
50M
5000
10000
20000
50000
y - Кількість порівнянь
x - Кількість елементів у масиві


Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить випадкову послідовність елементів)
1.2B
1B
800M
600M
400M
200M
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві


Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить випадкову послідовність елементів)
1.2B
200M
50M
5000
10000
20000
50000
y - Кількість порівнянь
x - Кількість елементів у масиві

Графік залежності часових характеристик оцінювання від розмірності масиву




Асимптотична оцінка гіршого випадку
y - Час сортування
x - Кількість елементів у масиві


Асимптотична оцінка кращого випадку
y - Час сортування
x - Кількість елементів у масиві

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

Таблиця 3.1 – Аналіз алгоритму на відповідність властивостям

Властивість	**Сортування гребінцем**
Стійкість	Ні, алгоритм не є стійким
«Природність» поведінки (Adaptability)	Так, алгоритм є природним
Базуються на порівняннях	Так, алгоритм базується на порівняннях
Необхідність в додатковій пам'яті (об'єм)	Ні, алгоритм не потребує додаткової пам’яті
Необхідність в знаннях про структури даних	Так, для використання цього алгоритму потрібно мати базові знання про структури даних

3.6 Псевдокод алгоритму

Підпрограма comb_sort(array):
Початок
    step := array.size()
    is_swap := True
    factor := 1.25
    повторити поки step > 1 **або** is_swap == True
        step := max(1, int(step / factor))
        is_swap := False
        повторити для i **від** 0 **до** list_to_sort.array () - step
            j := i+ step
            якщо list_to_sort[i] > list_to_sort[j]
                то

temp := list_to_sort[i]

list_to_sort[i] := list_to_sort[j]

list_to_sort[j] := temp

все повторити

все повторити

Кінець

3.7 Аналіз часової складності

Найгірша швидкодія алгоритму: O(n^2)

Найкраща швидкодія: O(n)

Середня швидкодія: O(n * log(n))

3.8 Програмна реалізація алгоритму

```python
import random


def comb_sort(list_to_sort: list[int]):
    # крок
    step = len(list_to_sort)
    # чи змінені елементи
    is_swap = True
    # коефіцієнт звужування
    factor = 1.25

    while step > 1 or is_swap:
        step = max(1, int(step / factor))
        is_swap = False

        for i in range(len(list_to_sort) - step):
            j = i + step
            if list_to_sort[i] > list_to_sort[j]:
                temp = list_to_sort[i]
                list_to_sort[i] = list_to_sort[j]
                list_to_sort[j] = temp


n = 10
```

```
25
26      random_list = random.sample(range(1, n+1), n)
27      print(random_list)
28
29      comb_sort(random_list)
30      print(random_list)
```

3.8.1 Вихідний код

```
import random


def comb_sort(list_to_sort: list[int]):
    # крок
    step = len(list_to_sort)
    # чи змінені елементи
    is_swap = True
    # коефіцієнт звужування
    factor = 1.25
    while step > 1 or is_swap:
        step = max(1, int(step / factor))
        is_swap = False


        for i in range(len(list_to_sort) - step):
            j = i + step
            if list_to_sort[i] > list_to_sort[j]:
                temp = list_to_sort[i]
                list_to_sort[i] = list_to_sort[j]
                list_to_sort[j] = temp


n = 10


random_list = random.sample(range(1, n+1), n)
print(random_list)
```

```
comb_sort(random_list)
print(random_list)
```

3.8.1 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

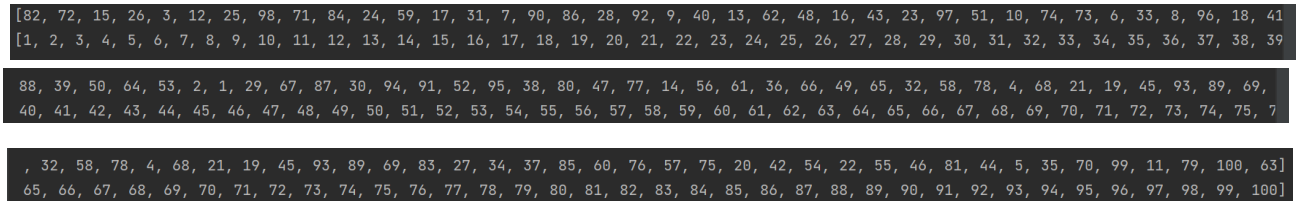
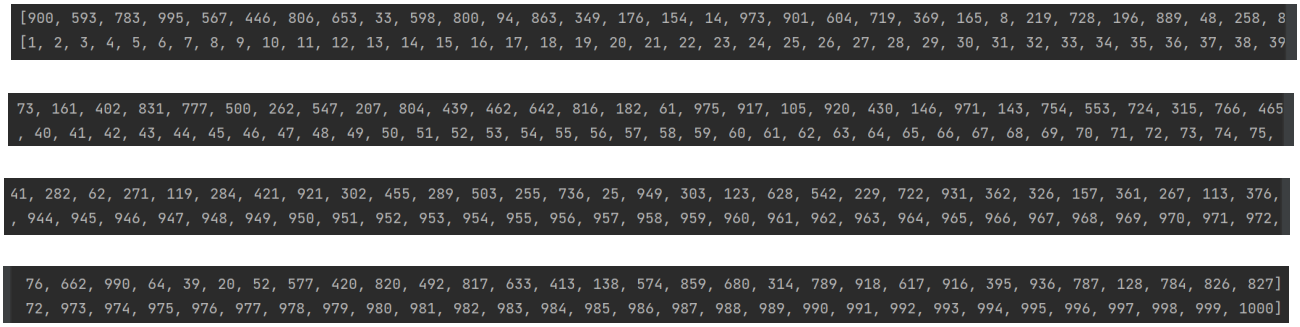


Рисунок 3.2 – Сортування масиву на 1000 елементів



3.9 Тестування алгоритму

Як було видно у попередньому пункті (п. 3.8.2), алгоритм правильно відсортував випадково згенерований масив на 100 і 1000 елементів, які не повторювались. Додатково протестуємо алгоритм сортування гребінцем на випадково згенерованому масиві з розмірністю 20 елементів (елементи можуть повторюватись).

Код:

```python
import random


def comb_sort(list_to_sort: list[int]):
    # крок
    step = len(list_to_sort)
    # чи змінені елементи
    is_swap = True
    # коефіцієнт звужування
    factor = 1.25

    while step > 1 or is_swap:
        step = max(1, int(step / factor))
        is_swap = False

        for i in range(len(list_to_sort) - step):
            j = i + step
            if list_to_sort[i] > list_to_sort[j]:
                temp = list_to_sort[i]
                list_to_sort[i] = list_to_sort[j]
                list_to_sort[j] = temp


n = 20
```

```
25
26      random_list = [random.randint(1, n) for i in range(n)]
27      print(random_list)
28
29      comb_sort(random_list)
30      print(random_list)
31
```

Результат тестування:

```
[20, 6, 10, 16, 19, 1, 2, 6, 13, 10, 7, 10, 4, 19, 3, 20, 8, 20, 7, 10]
[1, 2, 3, 4, 6, 6, 7, 7, 8, 10, 10, 10, 10, 13, 16, 19, 19, 20, 20, 20]
```

Як видно на скриншоті, алгоритм коректно відсортував випадково згенерований масив. Отже, наш алгоритм сортування гребінцем працює правильно.

3.9.1 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	9	0
100	99	0
1000	999	0
5000	4 999	0
10000	9 999	0
20000	19 999	0
50000	49 999	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	37	9
100	1 230	110
1000	22 057	1 514
5000	145 061	9 434
10000	320 072	19 740
20000	700 093	42 174
50000	2 000 059	110 230

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

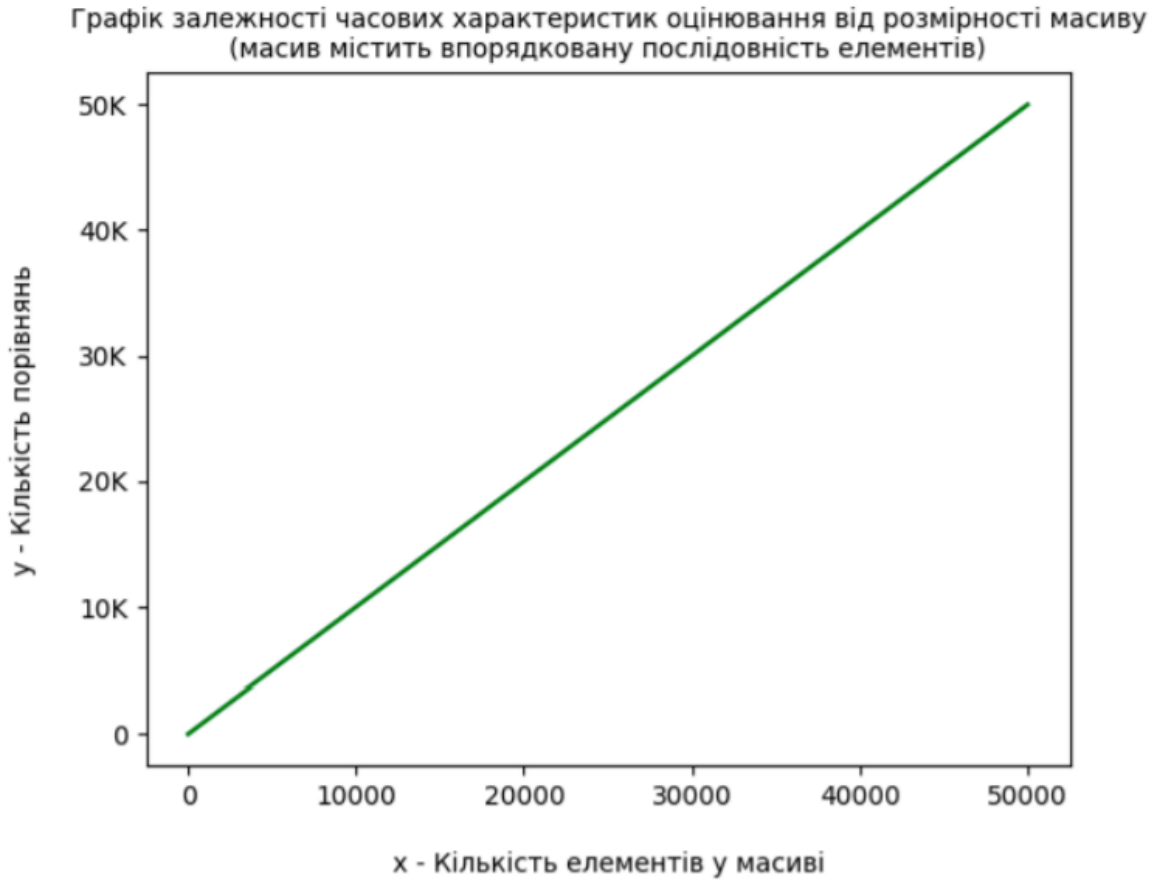
Таблиця 3.4 – Характеристика оцінювання алгоритму сортування гребінцем для випадкової послідовності елементів у масиві.

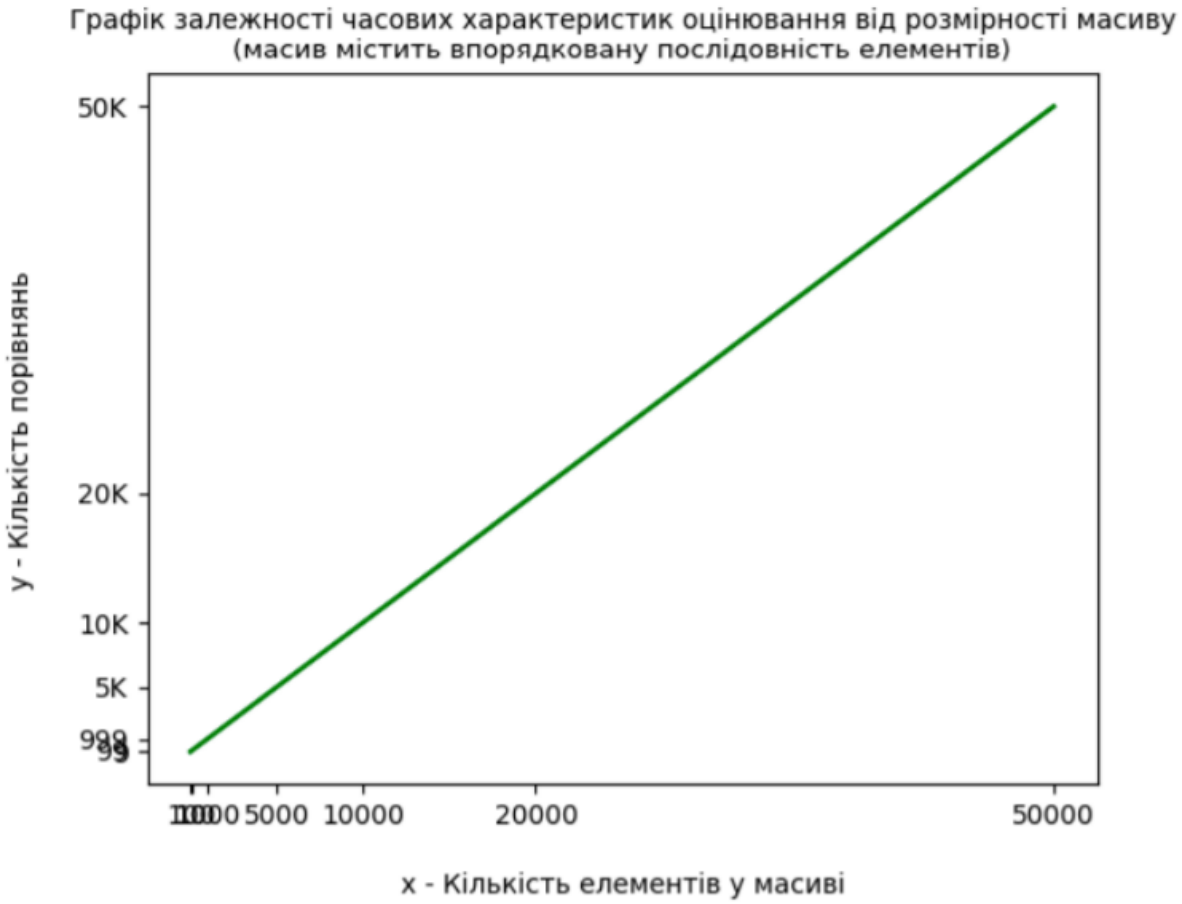
Розмірність масиву	Число порівнянь	Число перестановок
10	37	9
100	1 231	248
1000	22 057	4 214
5000	145 062	27 563
10000	320 072	60 961
20000	700 093	132 873
50000	2 000 062	368 885

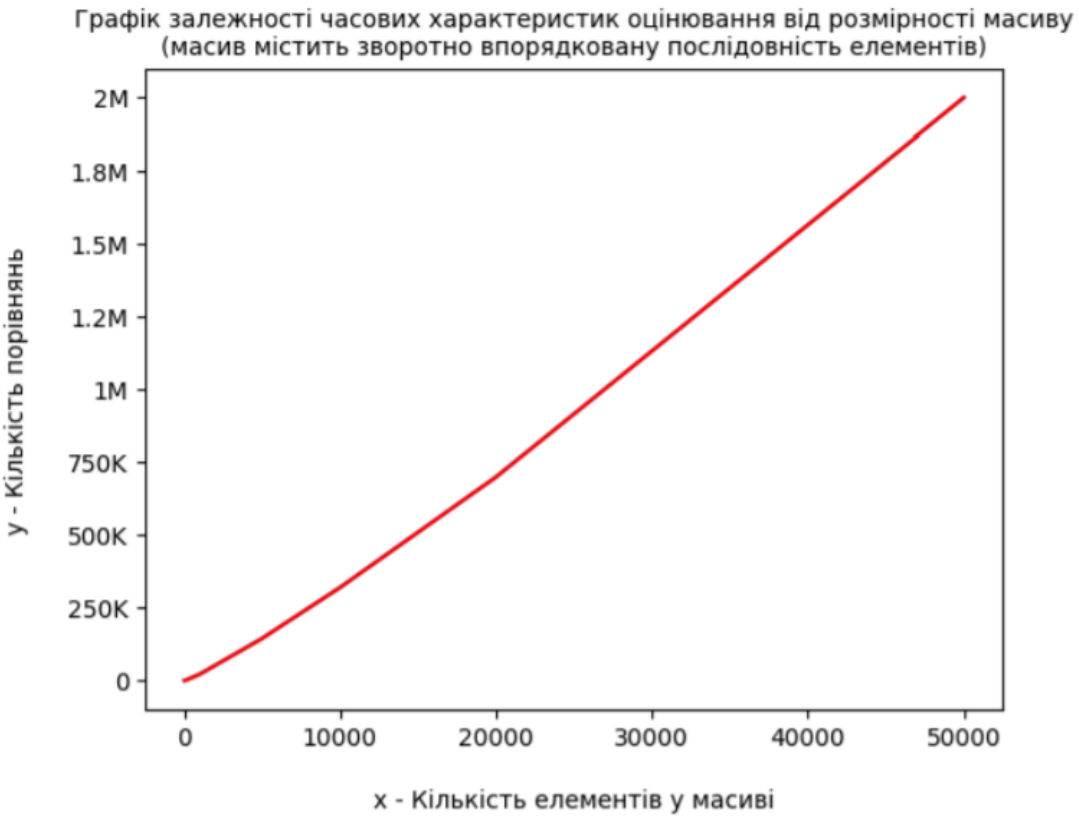
3.9.2 Графіки залежності часових характеристик оцінювання від розмірності масиву

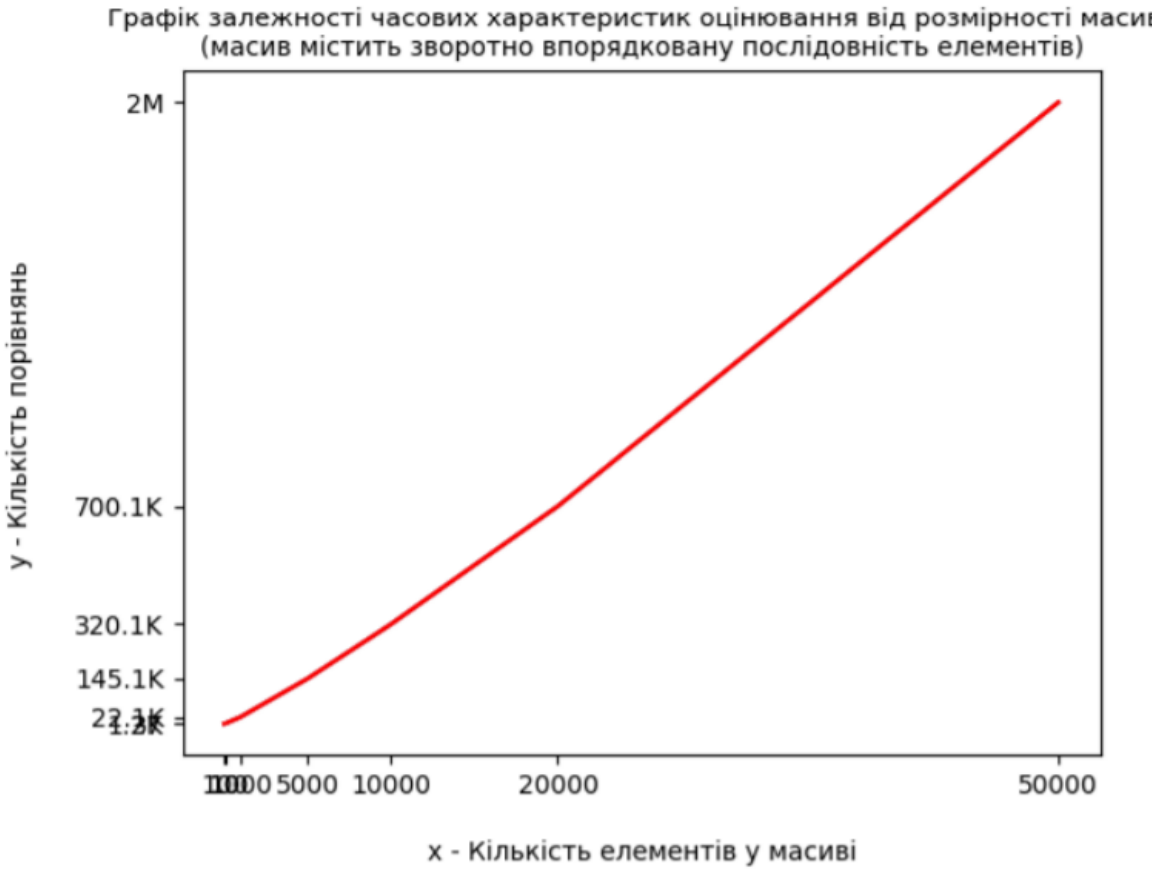
На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву для випадків, коли масиви містять упорядковану послідовність елементів (зелений графік), коли масиви містять зворотно упорядковану послідовність елементів (червоний графік), коли масиви містять випадкову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

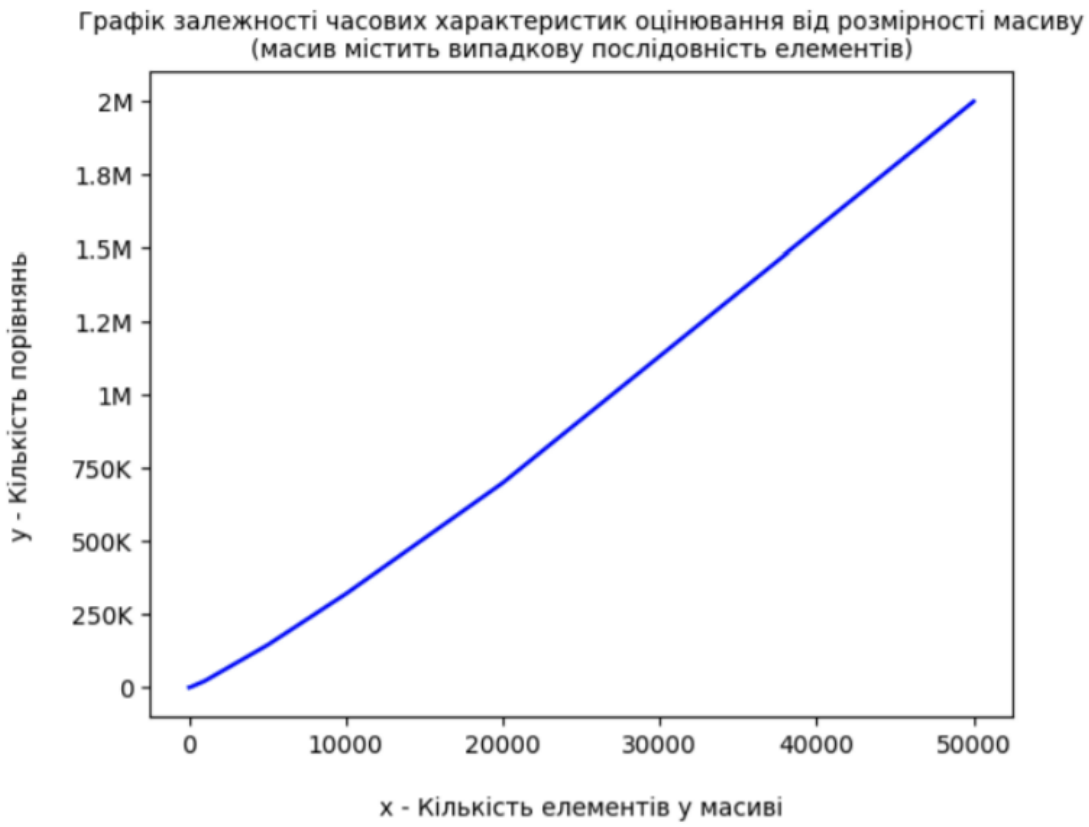
Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

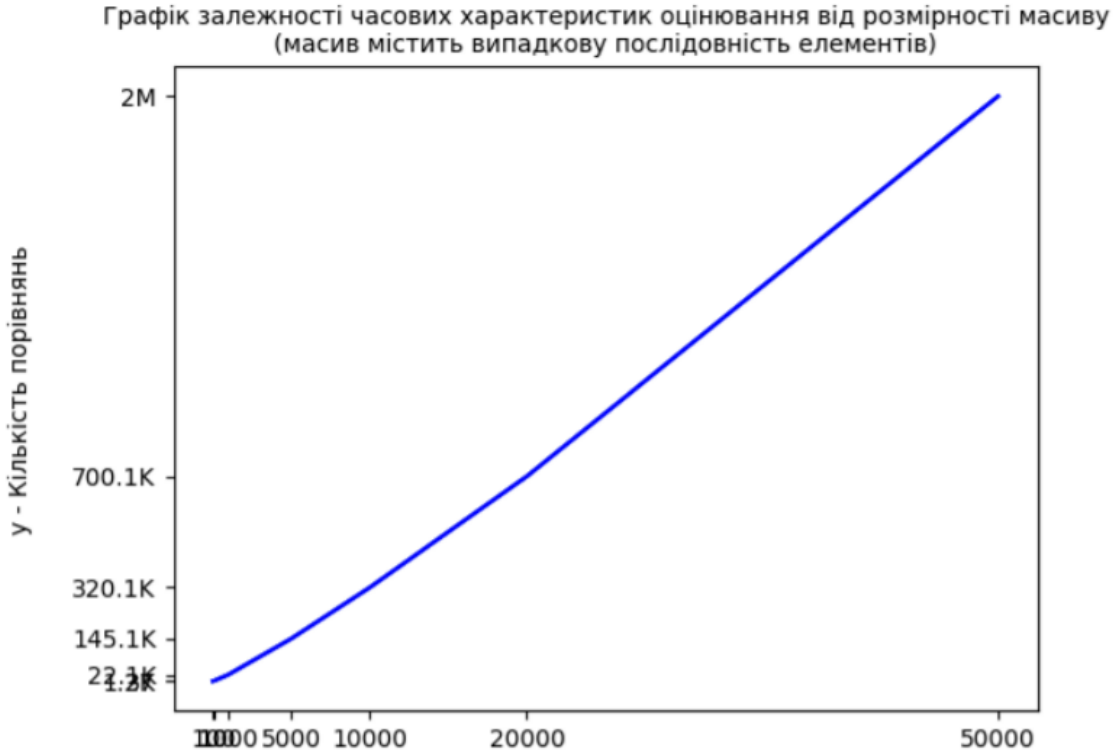



Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить впорядковану послідовність елементів)
50K
20K
10K
5K
y - Кількість порівнянь
5000
10000
20000
50000
x - Кількість елементів у масиві

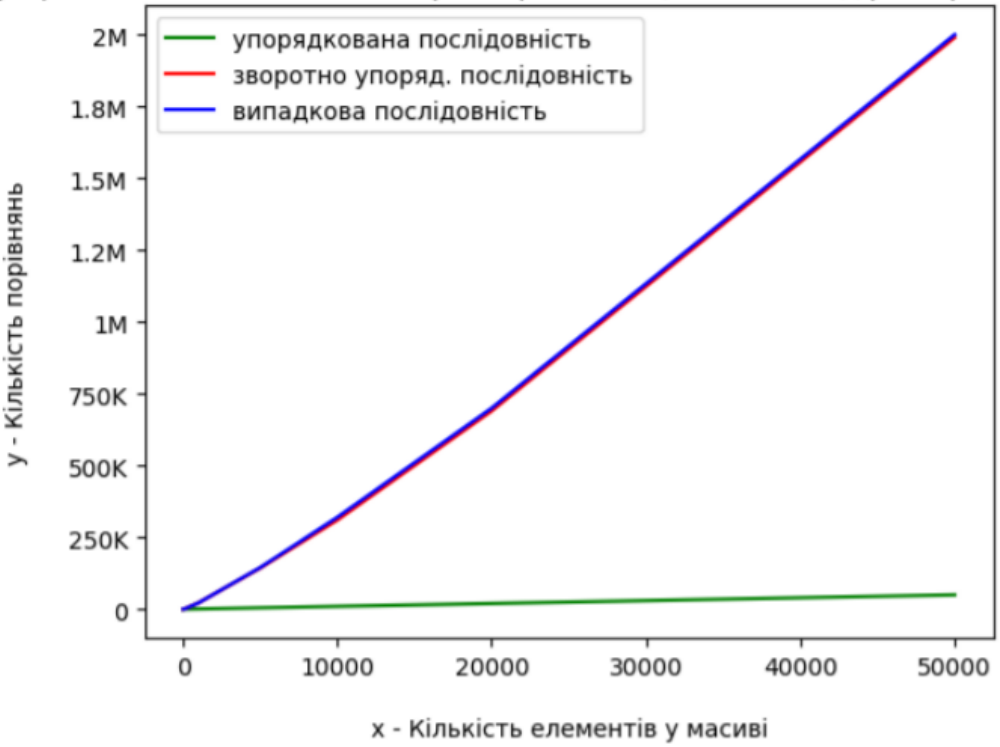

Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить зворотно впорядковану послідовність елементів)
2M
1.8M
1.5M
1.2M
1M
750K
500K
250K
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві

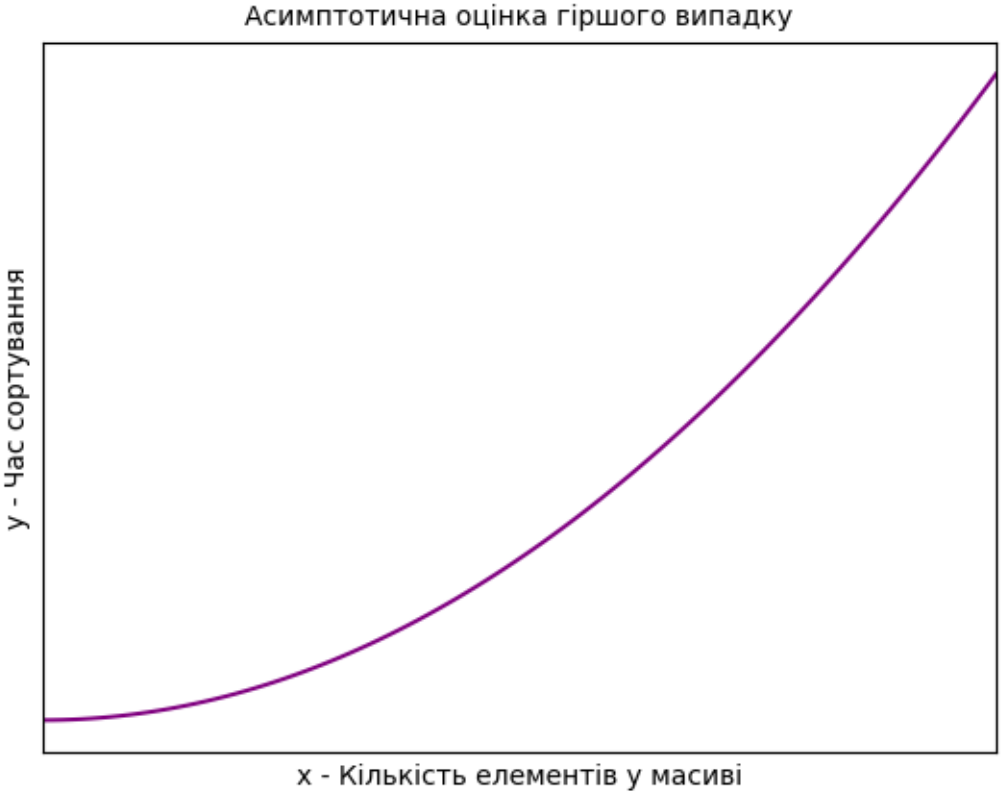

Графік залежності часових характеристик оцінювання від розмірності масив
(масив містить зворотно впорядковану послідовність елементів)
2M
700.1K
320.1K
145.1K
5000
10000
20000
50000
y - Кількість порівнянь
x - Кількість елементів у масиві

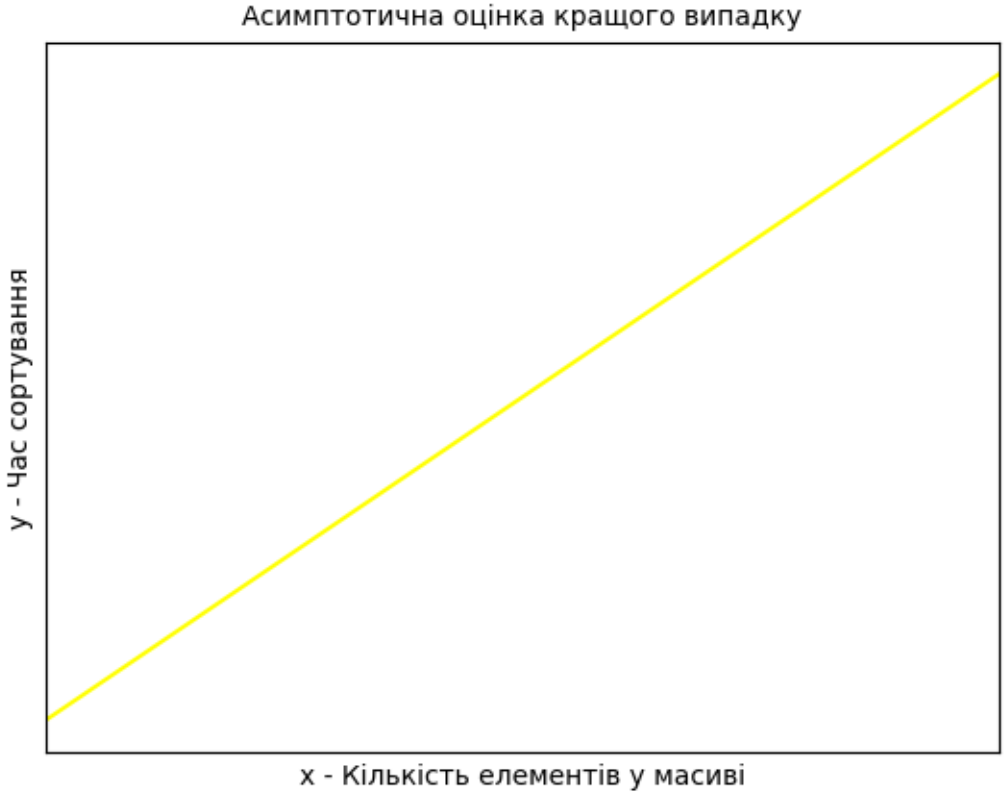

Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить випадкову послідовність елементів)
y - Кількість порівнянь
2M
1.8M
1.5M
1.2M
1M
750K
500K
250K
0
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві


Графік залежності часових характеристик оцінювання від розмірності масиву
(масив містить випадкову послідовність елементів)
y - Кількість порівнянь
2M
700.1K
320.1K
145.1K
5000
10000
20000
50000

Графік залежності часових характеристик оцінювання від розмірності масиву




Асимптотична оцінка гіршого випадку
y - Час сортування
x - Кількість елементів у масиві


Асимптотична оцінка кращого випадку
y - Час сортування
x - Кількість елементів у масиві

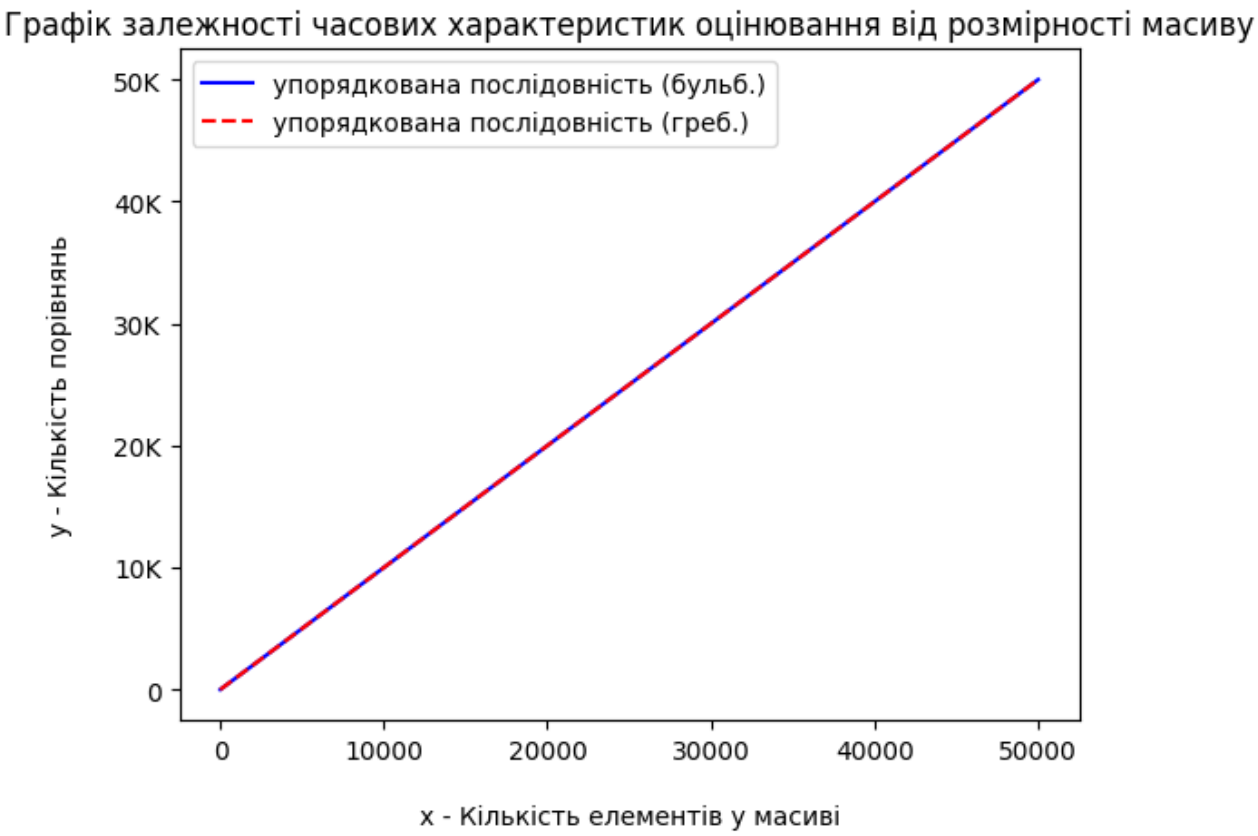
ВИСНОВОК

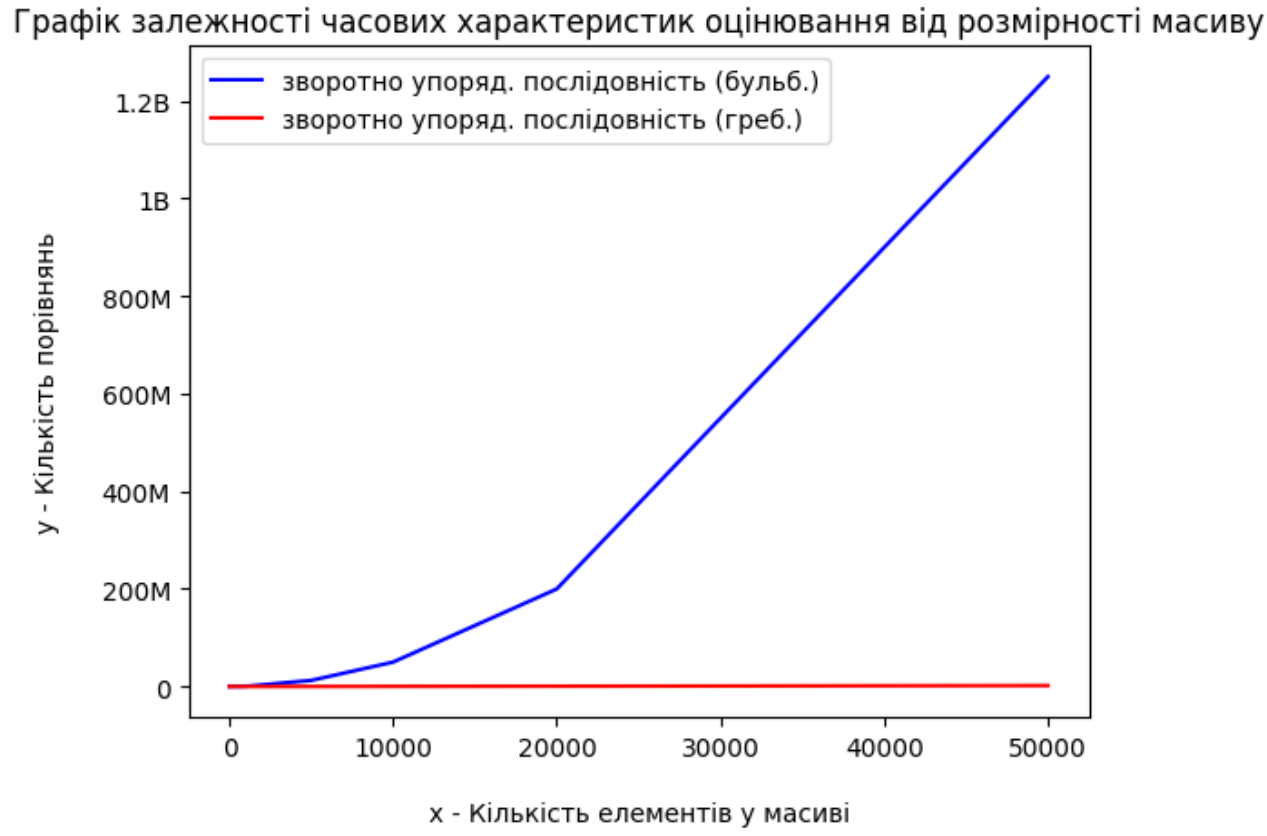
При виконанні даної лабораторної роботи я вивчив основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування і оцінив поріг їх ефективності.

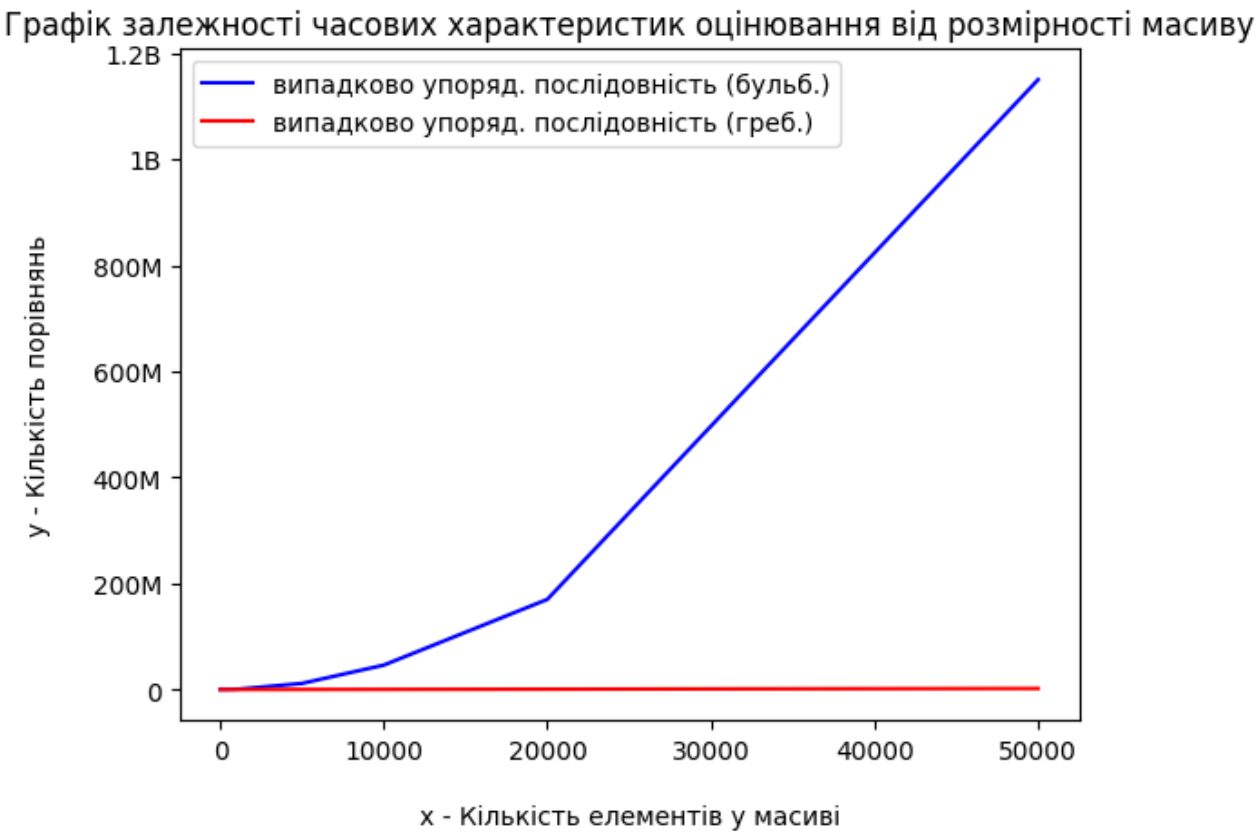
При виконанні порівняльного аналізу алгоритмів сортування бульбашкою і гребінцем, я зробив висновок, що сортування гребінцем є більш швидким алгоритмом. Переконатися у цьому можна, подивившись на графіки залежності часових характеристик оцінювання від розмірності масиву.

Як бачимо, для сортування масивів (особливо великих), алгоритму сортування бульбашкою потрібно виконати значну кількість порівнянь і перестановок (наприклад, на зворотно упорядкованому масиві з 20 000 елементів було виконано 199 990 000 порівнянь, а на зворотно упорядкованому масиві з 50 000 елементів було виконано 1 249 975 000 порівнянь).

А алгоритму сортування гребінцем, для сортування масивів (особливо великих) потрібно виконати значно менше порівнянь (наприклад, на зворотно упорядкованому масиві з 20 000 елементів було виконано 700 093 порівнянь, а на зворотно упорядкованому масиві з 50 000 елементів було виконано 2 000 059 порівнянь)


Графік залежності часових характеристик оцінювання від розмірності масиву
упорядкована послідовність (бульб.)
упорядкована послідовність (греб.)
50K
40K
30K
20K
10K
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві


Графік залежності часових характеристик оцінювання від розмірності масиву
зворотно упоряд. послідовність (бульб.)
зворотно упоряд. послідовність (греб.)
1.2B
1B
800M
600M
400M
200M
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві


Графік залежності часових характеристик оцінювання від розмірності масиву
випадково упоряд. послідовність (бульб.)
випадково упоряд. послідовність (греб.)
1.2B
1B
800M
600M
400M
200M
0
y - Кількість порівнянь
0
10000
20000
30000
40000
50000
x - Кількість елементів у масиві

КРИТЕРІЇ ОЦІНЮВАННЯ

У випадку здачі лабораторної роботи до 21.02.2022 включно максимальний бал дорівнює – 5. Після 21.02.2022 – 28.02.2022 максимальний бал дорівнює – 2,5. Після 28.02.2022 робота не приймається

Критерії оцінювання у відсотках від максимального балу:

- аналіз алгоритму на відповідність властивостям – 10%;
- псевдокод алгоритму – 15%;
- аналіз часової складності – 25%;
- програмна реалізація алгоритму – 25%;
- тестування алгоритму – 20%;
- висновок – 5%.