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-24 Харечко Олександр Іванович*** 27.02.2023
(шифр, прізвище, ім'я, по батькові)

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

Київ 2023

Лабораторна робота №1

Мета: дослідити поведінку двох алгоритмів сортування: метод бульбашки та метод вставки.

Опис роботи:

У даній роботі досліджуються три методи сортування: метод бульбашки, покращений алгоритм бульбашки та метод включення. Хоча обидва методи мають однакову асимптотичну складність - O(n*n), де n - розмір вхідного масиву, проте вважається, що в реальності метод бульбашки працює достатньо повільніше за метод включення.

Завдання:

Виконати аналіз алгоритму внутрішнього сортування на відповідність наступним властивостям:

- 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.

Виконання

1. Аналіз алгоритму на відповідність властивостям

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

Таблиця 3.1 – Аналіз алгоритму на відповідність властивостям

Властивість	**Bubble sort**	**Insertion sort**
Стійкість	+	+
«Природність» поведінки (Adaptability)	+	+
Базуються на порівняннях	+	+
Необхідність в додатковій пам’яті (об’єм)	-	+ (для тільки для ключа)
Необхідність у знаннях про структури даних	+	+

2. Псевдокод алгоритму

1) Сортування бульбашкою

```
for i := 0, i < n - 1, i += 1
        for j := 0, j < n - i – j, j += 1
                if arr[j] > arr[j + 1]

                        than  swap  arr[j],
                arr[j+1]
                end if
        end repeat
```

2) Модифіковане сортування бульбашкою

```
для i := 0, i < n - 1, i += 1
повторити
        notChanged := 1
        для j := 0, j < n - i – j, j += 1
        повторити
                якщо arr[j] > arr[j + 1]
                то
                        обмін arr[j], arr[j+1]
                        notChanged := 0
                все якщо
        все повторити
        якщо notChanged:
        то
                break
        всеякщо
все повторити
```

3) Insertion sort

```
для i := 2, i < n, i += 1
повторити
        key := arr[i]
        j := i - 1
        поки j >= 0 та arr[j] > key
        повторити
                arr[j+1] := arr[j]
                j := j - 1
        все повторити
        arr[j+1] := key
все повторити
```

3. Аналіз часової складності

1) Сортування бульбашкою

Загальна часова складність алгоритму $O(n^2)$.

Найгірший випадок – масив відсортований в зворотньому порядку.
Найкращий – масив вже відсортований.

Часова складність найгіршого випадку:

- Кількість порівнянь (n – 1) * n/2
- Кількість обмінів (n – 1) * n/2

Часова складність найкращого випадку:

- Кількість порівнянь (n – 1) * n/2
- Кількість обмінів 0

2) Модифіковане сортування бульбашкою

Загальна часова складність алгоритму $O(n^2)$. Найгірший випадок – масив відсортований навпаки, найкращий – масив вже відсортований.
Часова складність найгіршого випадку:

- Кількість порівнянь (n – 1) * n/2
- Кількість обмінів (n – 1) * n/2 Часова

складність найкращого випадку:

- Кількість порівнянь n – 1
- Кількість обмінів 0

3) Insertion sort

Загальна часова складність алгоритму $O(n^2)$. Найгірший випадок – масив відсортований навпаки, найкращий – масив вже відсортований.
Часова складність найгіршого випадку:

- Кількість порівнянь (n – 1) * n/2
- Кількість перестановок (n – 1) * n/2

Часова складність найкращого випадку:

- Кількість порівнянь n – 1
- Кількість перестановок 0

4. Програмна реалізація алгоритму

1) Сортування бульбашкою

```
def bubble_sort(list_a):
    number_of_comparison = 0
    number_of_permutation = 0

    for i in range(len(list_a) - 1):

        for j in range(len(list_a) - i - 1):

            number_of_comparison += 1
            if list_a[j] > list_a[j + 1]:
                number_of_permutation += 1
                list_a[j], list_a[j + 1] = list_a[j + 1], list_a[j]
    return list_a, number_of_comparison, number_of_permutation
```

2) Модифіковане сортування бульбашкою

```
def upgraded_bubble_sort(array):
    number_of_comparison = 0
    number_of_permutation = 0
    for i in range(len(array)):

        swapped = False

        for j in range(0, len(array) - i - 1):
            number_of_comparison += 1

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

```
                number_of_permutation += 1
                array[j], array[j + 1] = array[j + 1], array[j]
                swapped = True

        if swapped == False:
            break

    return array, number_of_comparison, number_of_permutation
```

3) Insertion sort

```
def insertion_sort(array):
    number_of_comparison = 0
    number_of_permutation = 0
    for i in range(1, len(array)):

        key = array[i]

        j = i - 1
        while j >= 0 and key < array[j]:
            number_of_comparison += 1
            array[j + 1] = array[j]
            number_of_permutation += 1
            j -= 1
        array[j + 1] = key

    return array, number_of_comparison, number_of_permutation
```

Вихідний код

```
from random import uniform
from time import perf_counter
import matplotlib.pyplot as plt

def main():
    size = (10, 100, 1000)
    for i in size:
        a = [round(uniform(0, 100), 4) for _ in range(i)]
        text_info(a, i)
        plot_building(i, a)


def plot_building(size, a):
    plt.title('Bubble sort')
    plt.ylabel('Operations')
    plt.xlabel('Array size')
    x_axis = []
    y_axis = [[], [], []]

    for i in range(1, size+1):
        arr = gen_arr(i)
        x_axis.append(i)
        y_axis[0].append(bubble_sort(arr[:])[1]+bubble_sort(arr[:])[2])

y_axis[1].append(bubble_sort(sorted(arr[:]))[1]+bubble_sort(sorted(arr[:]))[2])
        y_axis[2].append(bubble_sort(sorted(arr[:],
reverse=True))[1]+bubble_sort(sorted(arr[:], reverse=True))[2])

    plt.plot(x_axis, y_axis[0], color='red')
    plt.plot(x_axis, y_axis[1], color='green')
    plt.plot(x_axis, y_axis[2], color='purple')
    plt.plot(x_axis, [i - 1 for i in range(1, size + 1)], color='yellow',
linewidth=2)
    # plt.plot(x_axis, [i ** 2 for i in range(1, size + 1)], color='blue')
    plt.legend(("randomized", "sorted", "reversed", "best variant", "worst variant"))
```

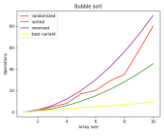
```
    plt.show()

    # plot for upgraded bubble sort
    plt.title('Upgraded Bubble sort')
    plt.ylabel('Operations')
    plt.xlabel('Array size')
    x_axis = []
    y_axis = [[], [], []]

    for i in range(1, size+1):
        arr = gen_arr(i)
        x_axis.append(i)

y_axis[0].append(upgraded_bubble_sort(arr[:])[1]+upgraded_bubble_sort(arr[:])[2])

y_axis[1].append(upgraded_bubble_sort(sorted(arr[:]))[1]+upgraded_bubble_sort(sorted(
arr[:]))[2])
        y_axis[2].append(upgraded_bubble_sort(sorted(arr[:],
reverse=True))[1]+upgraded_bubble_sort(sorted(arr[:], reverse=True))[2])

    plt.plot(x_axis, y_axis[0], color='red')
    plt.plot(x_axis, y_axis[1], color='green')
    plt.plot(x_axis, y_axis[2], color='purple')
    plt.plot(x_axis, [i - 1 for i in range(1, size + 1)], color='yellow',
linewidth=2)
    # plt.plot(x_axis, [i ** 2 for i in range(1, size + 1)], color='blue')
    plt.legend(("randomized", "sorted", "reversed", "best variant", "worst variant"))
    plt.show()

    # plot for insertion sort algorithm
    plt.title('Insertion sort')
    plt.ylabel('Operations')
    plt.xlabel('Array size')
    x_axis = []
    y_axis = [[], [], []]

    for i in range(1, size+1):
        arr = gen_arr(i)
        x_axis.append(i)
        y_axis[0].append(insertion_sort(arr[:])[1]+insertion_sort(arr[:])[2])

y_axis[1].append(insertion_sort(sorted(arr[:]))[1]+insertion_sort(sorted(arr[:]))[2])
        y_axis[2].append(insertion_sort(sorted(arr[:],
reverse=True))[1]+insertion_sort(sorted(arr[:], reverse=True))[2])

    plt.plot(x_axis, y_axis[0], color='red')
    plt.plot(x_axis, y_axis[1], color='green')
    plt.plot(x_axis, y_axis[2], color='purple')
    plt.plot(x_axis, [i - 1 for i in range(1, size + 1)], color='yellow',
linewidth=2)
    # plt.plot(x_axis, [i ** 2 for i in range(1, size + 1)], color='blue')
    plt.legend(("randomized", "sorted", "reversed", "best variant", "worst variant"))
    plt.show()

def text_info(a, i):
    print(f"Random generated array FOR LEN {i}: \n", i)
    start = perf_counter()
    print("Sorted array: ")
    # print(bubble_sort(a[:])[0], end='\n\n')

    print(f"Sorted time for Bubble function on random array: \n{perf_counter()-start}
sec")
    print("Number of comprasion: ", bubble_sort(a[:])[1])
    print("Number of permutation: ", bubble_sort(a[:])[2], end='\n\n')
    start = perf_counter()
    # print(bubble_sort(sorted(a[:]))[0])
    print(f"Sorted time for Bubble function on full-sorted array: \n{perf_counter()-
start} sec")
```

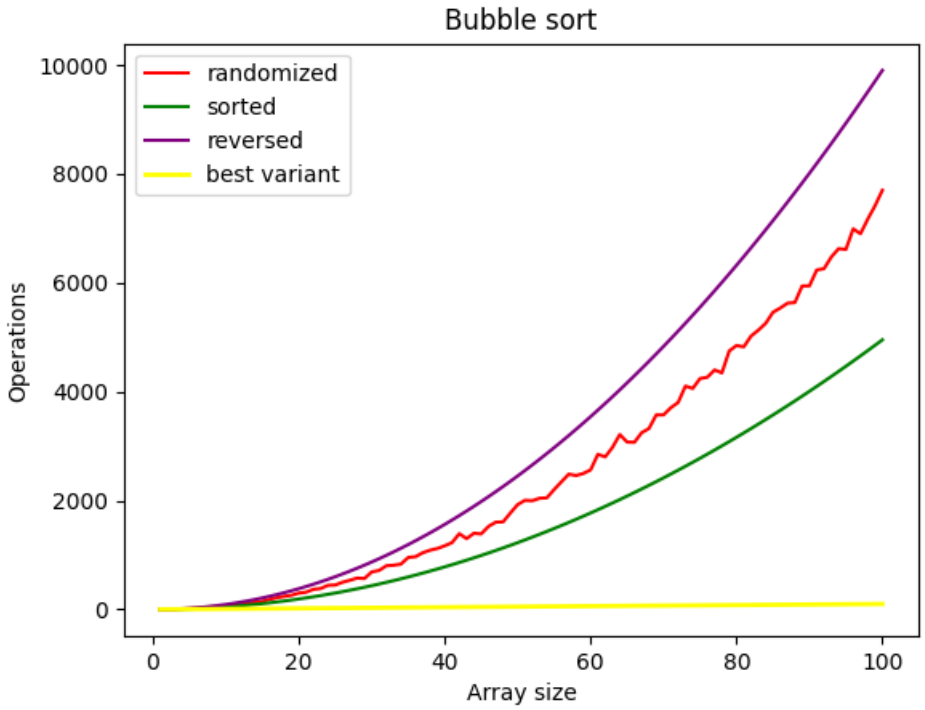
```
    print("Number of comprasion: ", bubble_sort(sorted(a[:]))[1])
    print("Number of permutation: ", bubble_sort(sorted(a[:]))[2], end='\n\n')
    start = perf_counter()
    # print(bubble_sort(sorted(a[:], reverse=True))[0])
    print(f"Sorted time for Bubble function on reversed-sorted array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", bubble_sort(sorted(a[:], reverse=True))[1])
    print("Number of permutation: ", bubble_sort(sorted(a[:], reverse=True))[2],
end='\n\n')

    print("\n-----------------------------------------\n")

    start = perf_counter()
    # print(upgraded_bubble_sort(a[:]))
    print(f"Sorted time for Upgraded Bubble function on random array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", upgraded_bubble_sort(a[:])[1])
    print("Number of permutation: ", upgraded_bubble_sort(a[:])[2], end='\n\n')
    start = perf_counter()
    # print(upgraded_bubble_sort(sorted(a[:]))[0])
    print(f"Sorted time for Upagraded Bubble sort function on full-sorted array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", upgraded_bubble_sort(sorted(a[:]))[1])
    print("Number of permutation: ", upgraded_bubble_sort(sorted(a[:]))[2],
end='\n\n')
    start = perf_counter()
    # print(upgraded_bubble_sort(sorted(a[:], reverse=True))[0])
    print(f"Sorted time for Upgraded Bubble sort function on reversed-sorted array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", upgraded_bubble_sort(sorted(a[:],
reverse=True))[1])
    print("Number of permutation: ", upgraded_bubble_sort(sorted(a[:],
reverse=True))[2], end='\n\n')

    print("\n-----------------------------------------\n")

    start = perf_counter()
    # print(insertion_sort(a[:]))
    print(f"Sorted time for Insertion sort function on random array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", insertion_sort(a[:])[1])
    print("Number of permutation: ", insertion_sort(a[:])[2], end='\n\n')
    start = perf_counter()
    # print(insertion_sort(sorted(a[:]))[0])
    print(f"Sorted time for Insertion sort function on full-sorted array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", insertion_sort(sorted(a[:]))[1])
    print("Number of permutation: ", insertion_sort(sorted(a[:]))[2], end='\n\n')
    start = perf_counter()
    # print(insertion_sort(sorted(a[:], reverse=True))[0])
    print(f"Sorted time for Insertion sort function on reversed-sorted array:
\n{perf_counter()-start} sec")
    print("Number of comprasion: ", insertion_sort(sorted(a[:], reverse=True))[1])
    print("Number of permutation: ", insertion_sort(sorted(a[:], reverse=True))[2],
end='\n\n')


def gen_arr(len):
    return [uniform(-10, 10) for _ in range(len)]
```

```
def bubble_sort(list_a):
    number_of_comparison = 0
    number_of_permutation = 0

    for i in range(len(list_a) - 1):

        for j in range(len(list_a) - i - 1):

            number_of_comparison += 1
            if list_a[j] > list_a[j + 1]:
                number_of_permutation += 1
                list_a[j], list_a[j + 1] = list_a[j + 1], list_a[j]
    return list_a, number_of_comparison, number_of_permutation


def upgraded_bubble_sort(array):
    number_of_comparison = 0
    number_of_permutation = 0
    for i in range(len(array)):

        swapped = False

        for j in range(0, len(array) - i - 1):
            number_of_comparison += 1

            if array[j] > array[j + 1]:
                number_of_permutation += 1
                array[j], array[j + 1] = array[j + 1], array[j]
                swapped = True

        if swapped == False:
            break

    return array, number_of_comparison, number_of_permutation


def insertion_sort(array):
    number_of_comparison = 0
    number_of_permutation = 0
    for i in range(1, len(array)):

        key = array[i]

        j = i - 1
        while j >= 0 and (number_of_comparison := number_of_comparison + 1) and
 key < array[j]:
            array[j + 1] = array[j]
            number_of_permutation += 1
            j -= 1
        array[j + 1] = key

    return array, number_of_comparison, number_of_permutation


if __name__ == "__main__":
    main()
```

Приклад роботи

На рисунках 3.1 і 3.2,3.3 показані приклади роботи програми для трьох функцій сортування для масивів на 10, 100 і 1000 елементів відповідно.





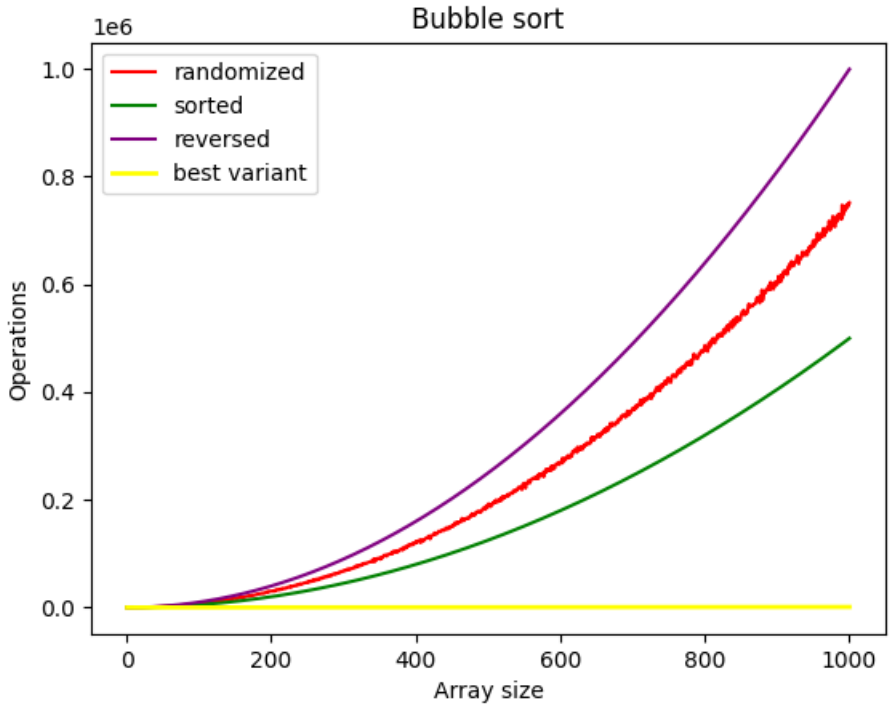
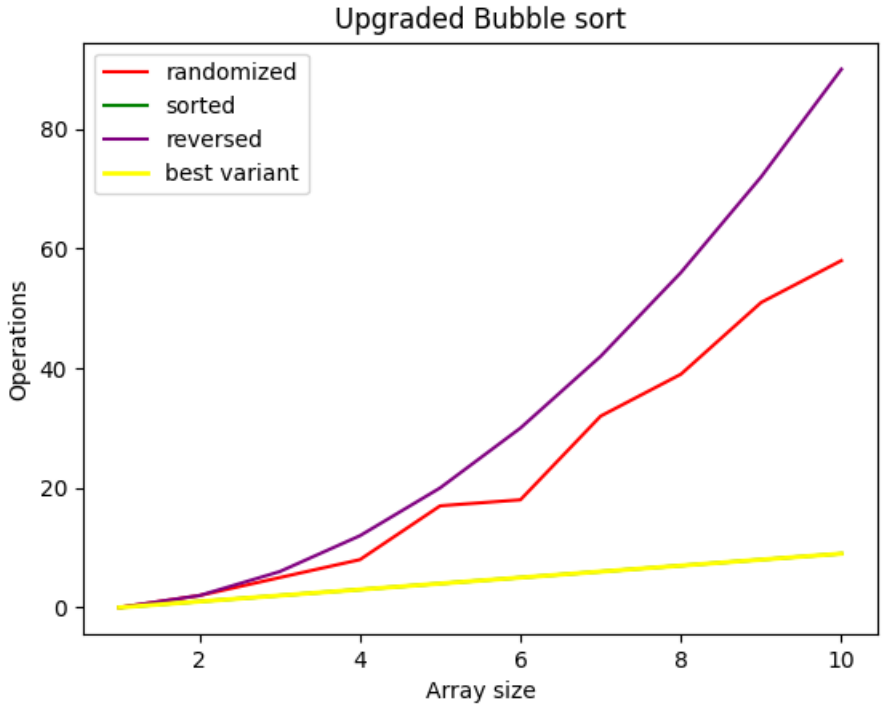


Рисунок 3.1 – Сортування масиву алгоритмом bubble sort



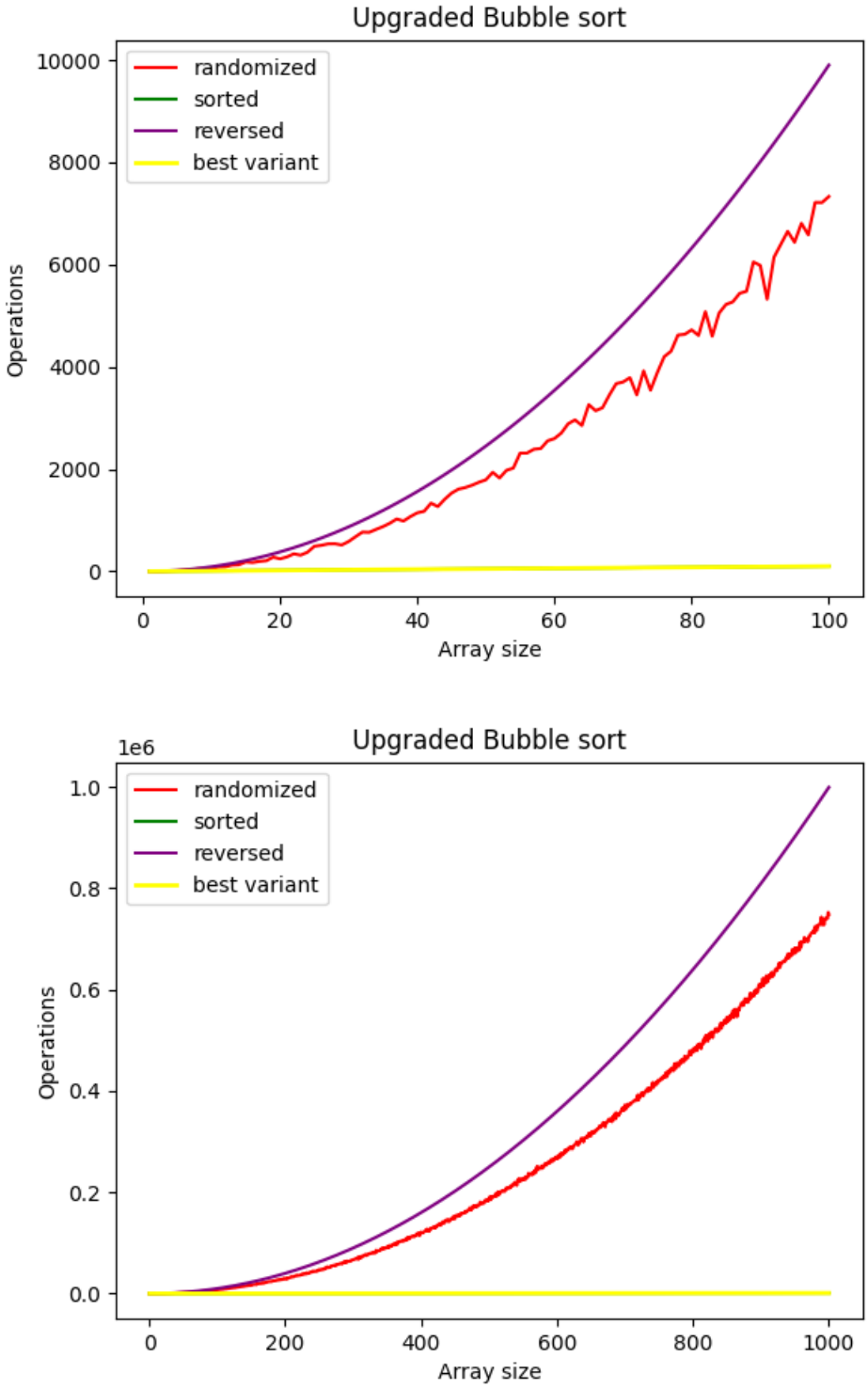
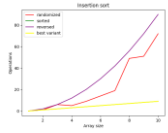
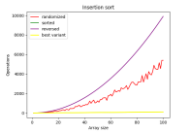


Рисунок 3.2 – Сортування масиву алгоритмом upgraded bubble sort


Insertion sort
randomized
sorted
reversed
best variant
Operations
0
20
40
60
80
2
4
6
8
10
Array size


Insertion sort
randomized
sorted
reversed
best variant
Operations
0
20000
40000
60000
80000
100000
0
20
40
60
80
100
Array size

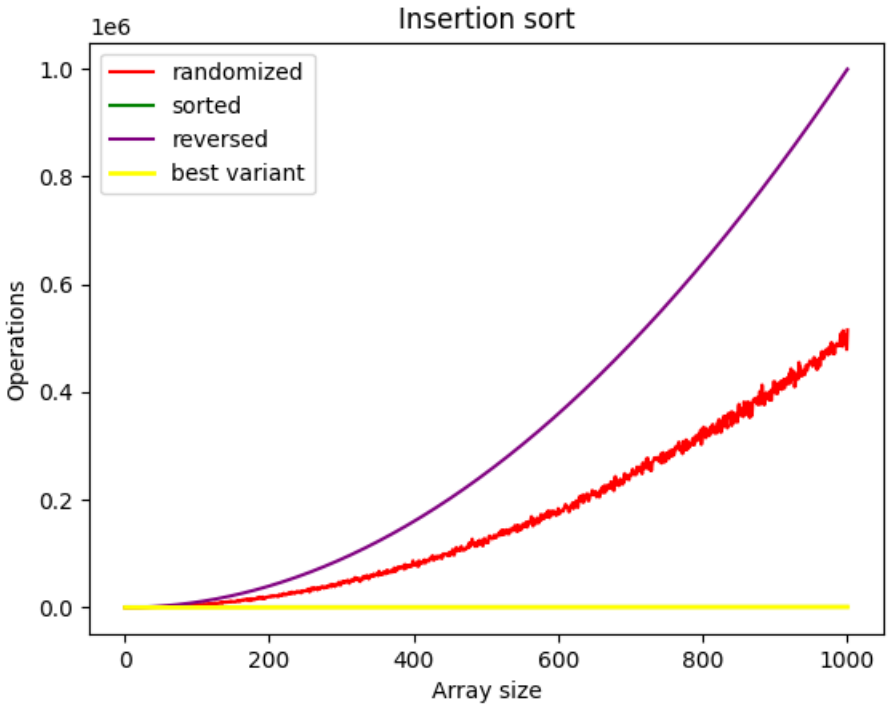


Рисунок 3.3 – Сортування масиву алгоритмом insertion sort

Тестування алгоритму
Часові характеристики оцінювання

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

Таблиця 3.2 – Характеристики оцінювання алгоритму сортування buble sort, upgraded buble sort та insertion sort для упорядкованої послідовності елементів у масиві.

Розмірність масиву	Число порівнянь	Число перестановок
10	45 9 9	0 0 0
100	4950 99 99	0 0 0
1000	499500 999 999	0 0 0
5000	12497500 4999 4999	0 0 0
10000	49995000 9999 9999	0 0 0
20000	199990000 19999 19999	0 0 0
50000	1249975000 49999 49999	0 0 0

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

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

Розмірність	Число порівнянь	Число перестановок
10	45 45 45	45 45 45
100	4950 4950 4950	4950 4950 4950
1000	499500 499500 499500	499500 499500 499500
5000	12497500 12497500 12497500	12497500 12497500 12497500
10000	49995000 49995000 49995000	49995000 49995000 49995000
20000	199990000 199990000 199990000	199990000 199990000 199990000
50000	1249975000 1249975000 124975000	1249975000 1249975000 124975000

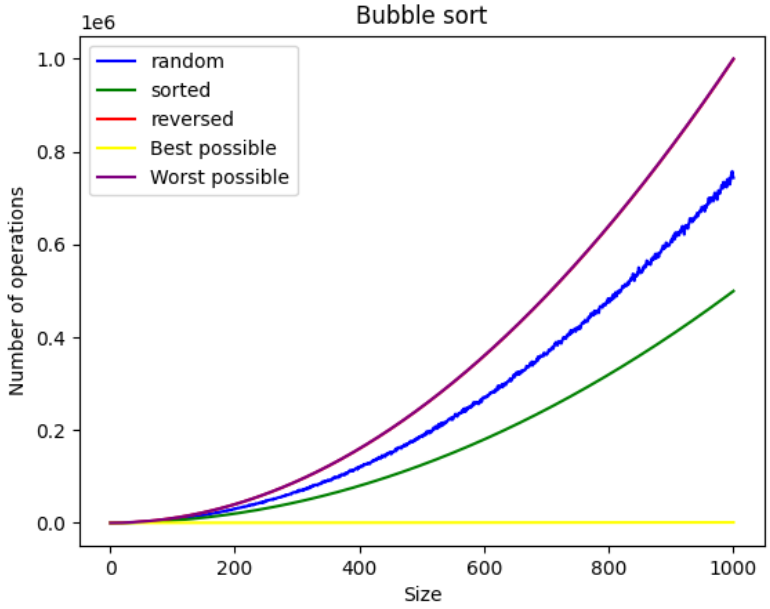
У таблиці 3.4 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування бульбашки, модифікованого алгоритму та Insertion_sort для масивів різної розмірності, масиви містять випадкову послідовність елементів.

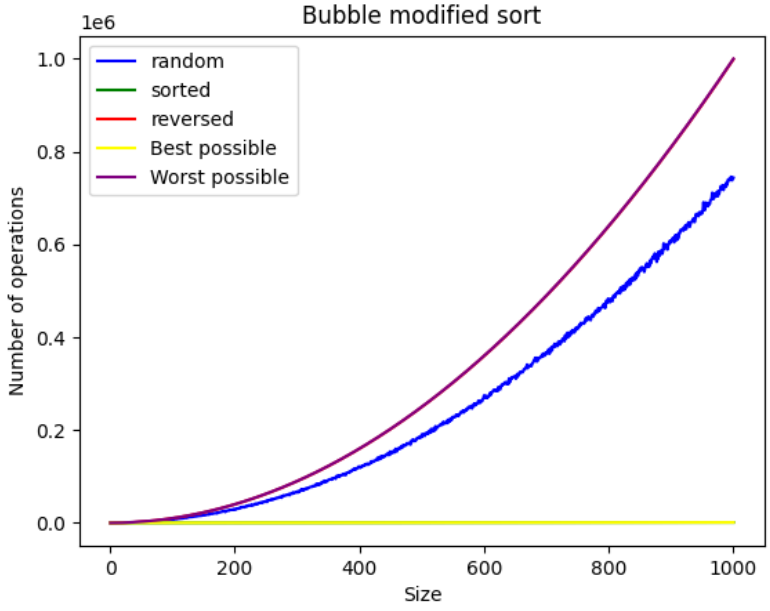
Таблиця 3.4 – Характеристика оцінювання алгоритму сортування бульбашки, модифікованого алгоритму та Insertion_sort для випадкової послідовності елементів у масиві.

Розмірність	Число порівнянь	Число перестановок
10	45 44 32	26 26 26
100	4950 4949 2818	2725 2725 2725
1000	499500 499247 246652	245659 245659 245659
5000	12497500 12497004 6295835	6290845 6290845 6290845
10000	49995000 49964865 24958691	24972046 24972046 24972046
20000	199990000 199984644 99483125	99871274 99871274 99871274
50000	1249975000 1249904005 628530999	628480578 628480578 628480578

3.1.1 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.





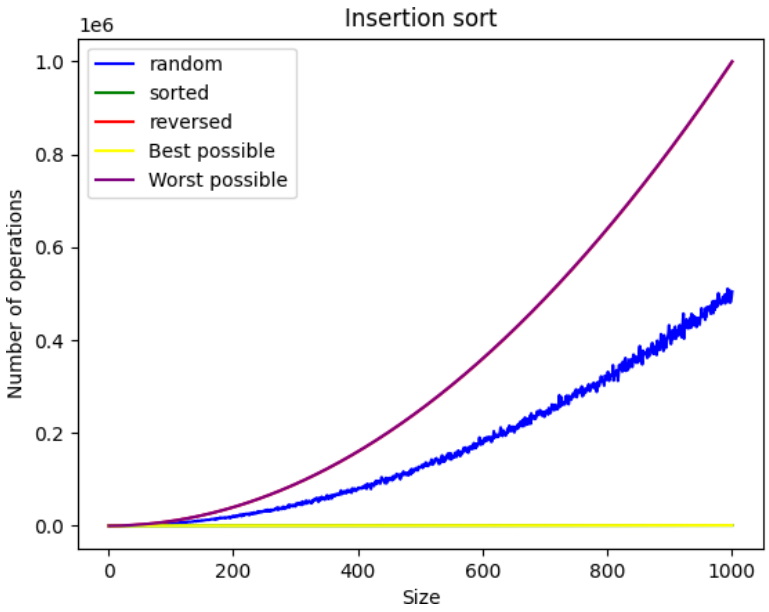


Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

ВИСНОВОК

На цій лабораторній роботі ми дослідили роботу алгоритмів сортування, а саме: сортування бульбашки, покращений алгоритм сортування бульбашки та сортування вставками. За ходом лабараторної роботи я дослідив час роботи кожного алгоритму та обчислив складність алгоритмів. В найгіршому випадку складність кожного з них є квадратичною $O(n^2)$. Я знайшов спосіб покращити сортування бульбашкою додавши перевірку на те, що масив вже відсортований, це дозволили значно зменшити час виконання алгоритму. Також я побудував графіки залежностей часових характеристик від розміру масиву, з яких можна наглядно бачити складності кожного з алгоритмів в конкретних випадках.