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 ІП-23 Зубарев Микола Костянтинович

Перевірив Соколовський В.В.

Київ 2023

ЗМІСТ

1. **МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3**
2. **ЗАВДАННЯ 4**
3. **ВИКОНАННЯ 5**

3.1 АНАЛІЗ АЛГОРИТМУ НА ВІДПОВІДНІСТЬ ВЛАСТИВОСТЯМ 5

3.2 ПСЕВДОКОД АЛГОРИТМУ..... 5
3.3 АНАЛІЗ ЧАСОВОЇ СКЛАДНОСТІ 6
3.4 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ 7
3.4.1 *Вихідний код 7*
3.4.2 *Приклад роботи 10*
3.5 ТЕСТУВАННЯ АЛГОРИТМУ 11
3.5.1 *Часові характеристики оцінювання 11*
3.5.2 Графіки залежності часових характеристик оцінювання від розмірності масиву 13

4. **ВИСНОВОК 15**
5. **КРИТЕРІЇ ОЦІНЮВАННЯ 15**

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	Модифікований алгоритм бульбашки
3	Insertion_sort

3. Виконання

3.1 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	Сортування бульбашкою	Insertion sort
Стійкість	+	+
«Природність» поведінки(Adaptability	-	+
Базуються на порівняннях	+	+
Необхідність в додатковій пам’яті(об’єм)	-	+
Необхідність в знаннях про структури даних	+	+

3.2 Псевдокод алгоритму

Псевдокод для сортування бульбашкою

```
def bubbleSort(num):
    compare, transition = 0, 0
    n = len(num)
    for i in range(n):
        for j in range(0, n - i - 1):
            compare += 1
            if num[j] > num[j + 1]:
                transition += 1
                num[j], num[j + 1] = num[j + 1], num[j]
    return compare, transition
```

Псевдокод для модифікованого сортування бульбашкою

```
def optimized_bubble(arr):
    compare = 0
    transposition = 0
    n = len(arr)
    for i in range(n):
        swapped = False
```

```
        for j in range(0, n-i-1):
            compare += 1
            if arr[j] > arr[j+1]:
                transposition += 1
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return compare, transposition
```

Псевдокод для сортування вставками

```
def insertion_sort(array):
    compare, transition = 0, 0
    for i in range(1, len(array)):
        compare += 1
        key = array[i]
        j = i-1
        while j >= 0 and key < array[j]:
            transition += 1
            array[j + 1] = array[j]
            compare += 1
            j -= 1
        array[j + 1] = key
    return compare, transition
```

3.3 Аналіз часової складності

Сортування бульбашкою

Часова складність сортування бульбашкою є O(n^2), де n - кількість елементів в масиві.

Перший цикл for виконується n разів, а внутрішній цикл for виконується n-1, n-2, ..., 1 разів в залежності від значення i, тому ми будемо множити n. Тому, усього виконується n + (n-1) + (n-2) + ... + 1 = n * (n + 1) / 2 викликів внутрішнього циклу. Таким чином, загальна часова складність сортування бульбашкою є O(n^2).

Модифікована бульбашка

Часова складність оптимізованого алгоритму бульбашкового сортування становить O(n^2) у найгіршому випадку і O(n) у найкращому випадку.

У найгіршому випадку елементи сортуються у зворотному порядку, і алгоритм повинен зробити n-1 проходів через масив, щоб відсортувати елементи, з n-1 порівняннями на кожному проході. Це призводить до часової складності O(n^2).

Однак, у найкращому випадку, елементи вже відсортовано, і алгоритм завершить роботу після першого проходу, оскільки прапорець "swapped" не буде встановлено у True. У цьому випадку часова складність становить O(n), оскільки потрібен лише один прохід через масив.

Insertion sort

Часова складність алгоритму сортування вставками становить O(n^2), де n - кількість елементів у вхідному масиві. Це пов'язано з тим, що для кожного елемента масиву алгоритм повинен перебрати всі елементи перед ним, щоб знайти його правильну позицію у відсортованому масиві. У найгіршому випадку це означає, що для останнього елемента алгоритм повинен виконати n-1 порівнянь і n-1 обмінів, що в сумі дає нам (n-1) + (n-2) + ... + 1 = n(n-1)/2 порівнянь і замін. Отже, часова складність складає O(n^2).

3.4 Програмна реалізація алгоритму

3.4.1 Вихідний код

```
import random
import matplotlib.pyplot as plt

def bubbleSort(num):
    compare, transition = 0, 0
    n = len(num)
    for i in range(n):
        for j in range(0, n - i - 1):
            compare += 1
            if num[j] > num[j + 1]:
                transition += 1
                num[j], num[j + 1] = num[j + 1], num[j]
    return compare, transition

def optimized_bubble(arr):
    compare = 0
    transposition = 0
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            compare += 1
            if arr[j] > arr[j+1]:
                transposition += 1
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break

    return compare, transposition

def insertion_sort(array):
    compare, transition = 0, 0
    for i in range(1, len(array)):
        compare += 1
        key = array[i]
        j = i-1
        while j >= 0 and key < array[j]:
            transition += 1
            array[j + 1] = array[j]
            compare += 1
```

```
            j -= 1
        array[j + 1] = key
    return compare, transition

sort_method = int(input('Enter 1 for ordinary bubble sort, 2 for optimized
bubble sort, 3 for insertion sort: '))
num = int(input('Enter number of elements: '))

def more_than():
    if num <= 10:
        print("\nBefore sort: ", l)
        print("Sorted ascending: ", l_best)
        print("Sorted descending: ", l_worst)

l = []
l_best = []
l_worst = []
y_1 = []
y_2 = []
y_3 = []
x_axis = []

match sort_method:
    case 1:
        for i in range(1, num + 1):
            l.clear()
            l_best.clear()
            l_worst.clear()
            for k in range(i):
                l.append(random.randint(-10000, 10000))
                l_best.append(k)
                l_worst.append(i-k)
                more_than()
            com1, trans1 = bubbleSort(l)
            com2, trans2 = bubbleSort(l_best)
            com3, trans3 = bubbleSort(l_worst)
            x_axis.append(i)
            y_1.append([com1 + trans1])
            y_2.append([com2 + trans2])
            y_3.append([com3 + trans3])

        print('\nComparisons in the random case:', com1, '\nSwaps in the random
case:', trans1)
        print('\nComparisons in the best case:', com2, '\nSwaps in the best
case:', trans2)
        print('\nComparisons in the worst case:', com3, '\nSwaps in the worst
case:', trans3)

        y_best_asymptote = [2*i for i in range(1, num + 1)]
        y_worst_asymptote = [i*i for i in range(1, num + 1)]

        plt.plot(x_axis, y_1, color='b', label='Random')
        plt.plot(x_axis, y_2, 'g', label='Best')
        plt.plot(x_axis, y_3, 'r', label='Worst')
        plt.plot(x_axis, y_best_asymptote, color='k', label='Best asymptote')
        plt.plot(x_axis, y_worst_asymptote, color='m', label='Worst asymptote')

        plt.xlabel('Array dimension')
        plt.ylabel('Number of operations')
        plt.title(f'Bubble sort for {num} elements')
        plt.legend()
        plt.show()

    case 2:
        for i in range(1, num + 1):
```

```
            l.clear()
            l_best.clear()
            l_worst.clear()
            for k in range(i):
                l.append(random.randint(-10000, 10000))
                l_best.append(k)
                l_worst.append(i-k)
                more_than()
            com1, trans1 = optimized_bubble(l)
            com2, trans2 = optimized_bubble(l_best)
            com3, trans3 = optimized_bubble(l_worst)
            x_axis.append(i)
            y_1.append([com1 + trans1])
            y_2.append([com2 + trans2])
            y_3.append([com3 + trans3])

        print('\nComparisons in the random case:', com1, '\nSwaps in the random
case:', trans1)
        print('\nComparisons in the best case:', com2, '\nSwaps in the best
case:', trans2)
        print('\nComparisons in the worst case:', com3, '\nSwaps in the worst
case:', trans3)

        y_best_asymptote = [i for i in range(1, num + 1)]
        y_worst_asymptote = [i*i for i in range(1, num + 1)]

        plt.plot(x_axis, y_1, color='b', label='Random')
        plt.plot(x_axis, y_2, 'g', label='Best')
        plt.plot(x_axis, y_3, 'r', label='Worst')
        plt.plot(x_axis, y_best_asymptote, color='k', label='Best asymptote')
        plt.plot(x_axis, y_worst_asymptote, color='m', label='Worst asymptote')

        plt.xlabel('Array dimension')
        plt.ylabel('Number of operations')
        plt.title(f'Optimized bubble sort for {num} elements')
        plt.legend()
        plt.show()

    case 3:
        for i in range(1, num + 1):
            l.clear()
            l_best.clear()
            l_worst.clear()
            for k in range(i):
                l.append(random.randint(-10000, 10000))
                l_best.append(k)
                l_worst.append(i-k)
                more_than()
            com1, trans1 = insertion_sort(l)
            com2, trans2 = insertion_sort(l_best)
            com3, trans3 = insertion_sort(l_worst)
            x_axis.append(i)
            y_1.append([com1 + trans1])
            y_2.append([com2 + trans2])
            y_3.append([com3 + trans3])

        print('\nComparisons in the random case:', com1, '\nSwaps in the random
case:', trans1)
        print('\nComparisons in the best case:', com2, '\nSwaps in the best
case:', trans2)
        print('\nComparisons in the worst case:', com3, '\nSwaps in the worst
case:', trans3)

        y_best_asymptote = [2*i-1 for i in range(1, num + 1)]
        y_worst_asymptote = [i*i for i in range(1, num + 1)]
```

```
        plt.plot(x_axis, y_1, color='b', label='Random')
        plt.plot(x_axis, y_2, 'g', label='Best')
        plt.plot(x_axis, y_3, 'r', label='Worst')
        plt.plot(x_axis, y_best_asymptote, color='k', label='Best asymptote')
        plt.plot(x_axis, y_worst_asymptote, color='m', label='Worst asymptote')

        plt.xlabel('Array dimension')
        plt.ylabel('Number of operations')
        plt.title(f'Insertion sort for {num} elements')
        plt.legend()
        plt.show()
```

3.4.2 Приклад роботи

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

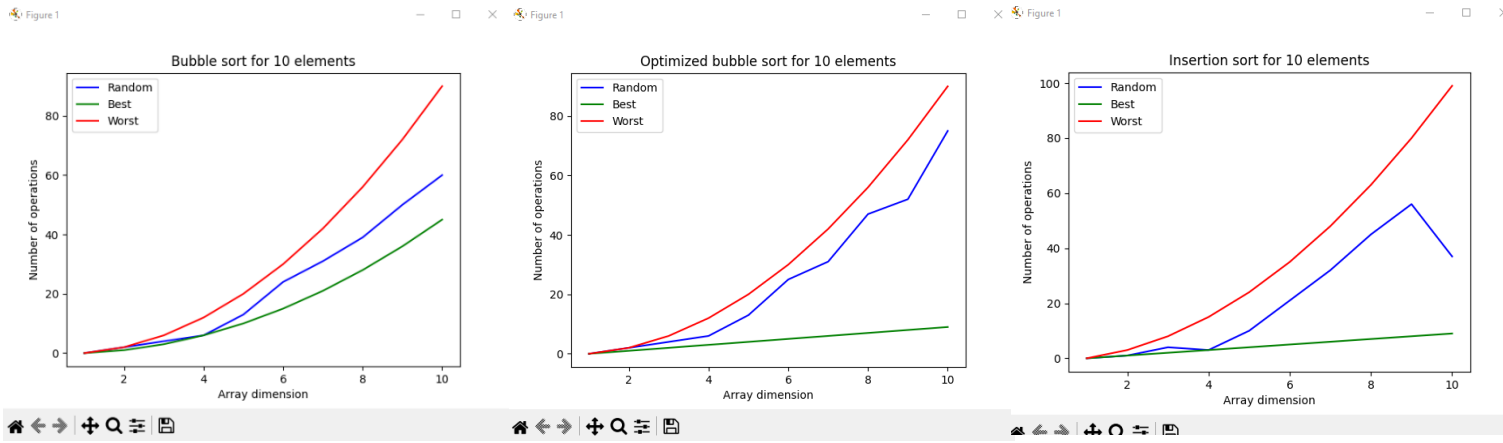


Рис. 3.1 – Сортування на 10 елементів

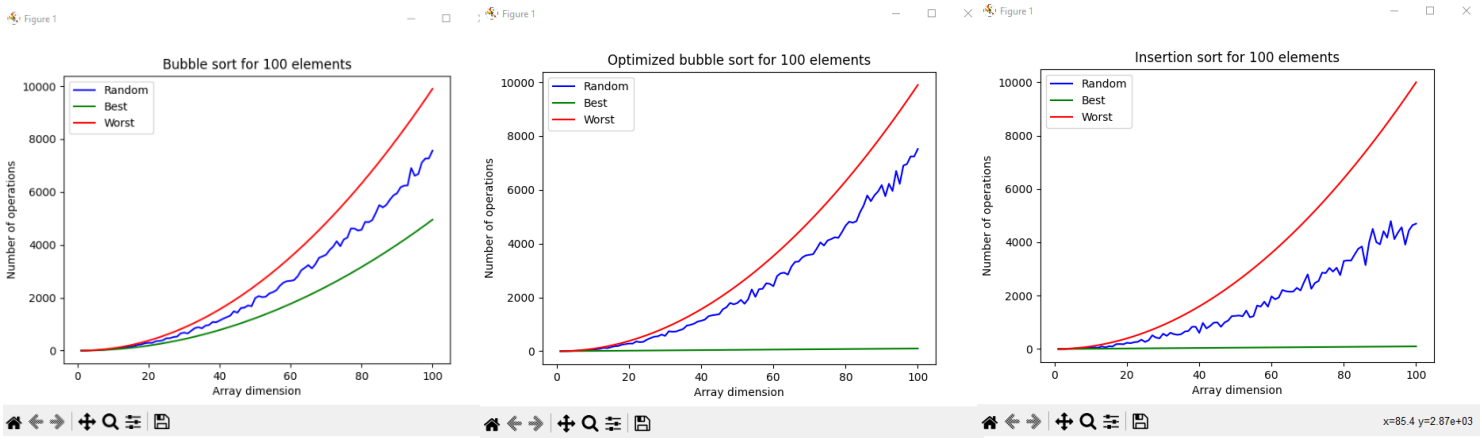


Рис 3.2 – Сортування на 100 елементів

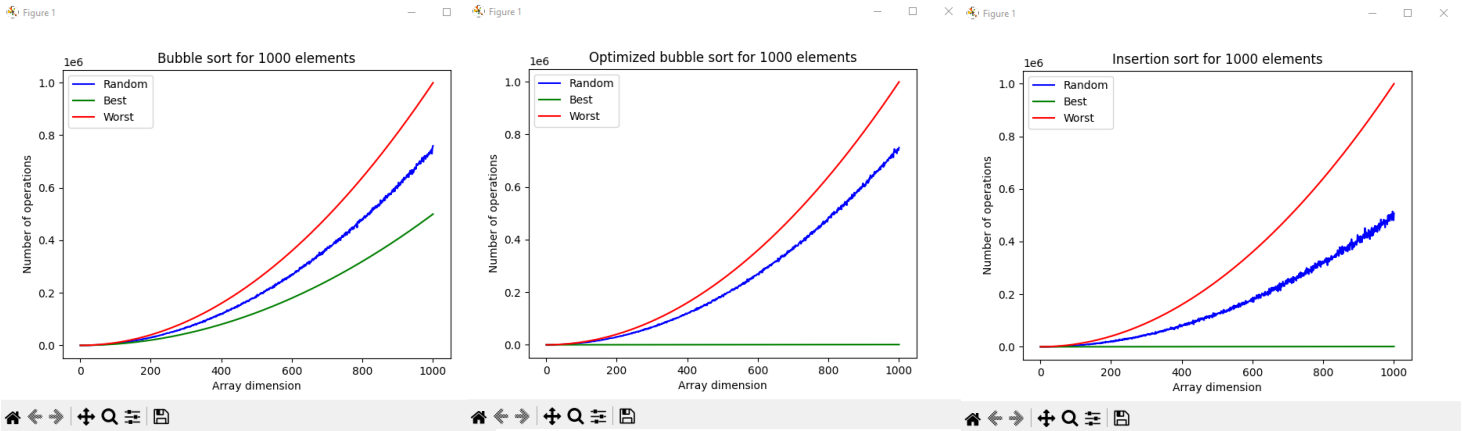


Рис 3.3 – Сортування на 1000 елементів

3.5 Тестування алгоритму

3.5.1 Часові характеристики оцінювання

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

Таблиця 3.2 – Характеристики оцінювання алгоритму сортування бульбашки, модифікованого алгоритму та Insertion_sort для упорядкованої послідовності елементів у масиві

Розмірність масиву	Число порівнянь Сортування; Модифіковане; Insertion			Число перестановок
10	45	9	9	0
100	4950	99	99	0
1000	49950	999	999	0
5000	12497500	4999	4999	0
10000	49995000	9999	9999	0
20000	199990000	19999	19999	0
50000	1249975000	49999	49999	0

Числа перестановок алгоритму сортування бульбашки, модифікованого алгоритму та вставкою 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 масиву	Число порівнянь Звичайне; Модифіковане; Вставкою		Число перестановок Звичайне; Модифіковане; Insertion		
10	45	54	45		
100	4950	5047	4944	4945	4948
1000	499500	500258	499270	449229	499259
5000	12497500	12500375	12495506	12495419	12495376
10000	49995000	50000403	49990399	49990471	49990404
20000	199990000	200000357	199980449	199980543	199980358
50000	1249975000	1250000656	1249950511	1249950506	1249950657

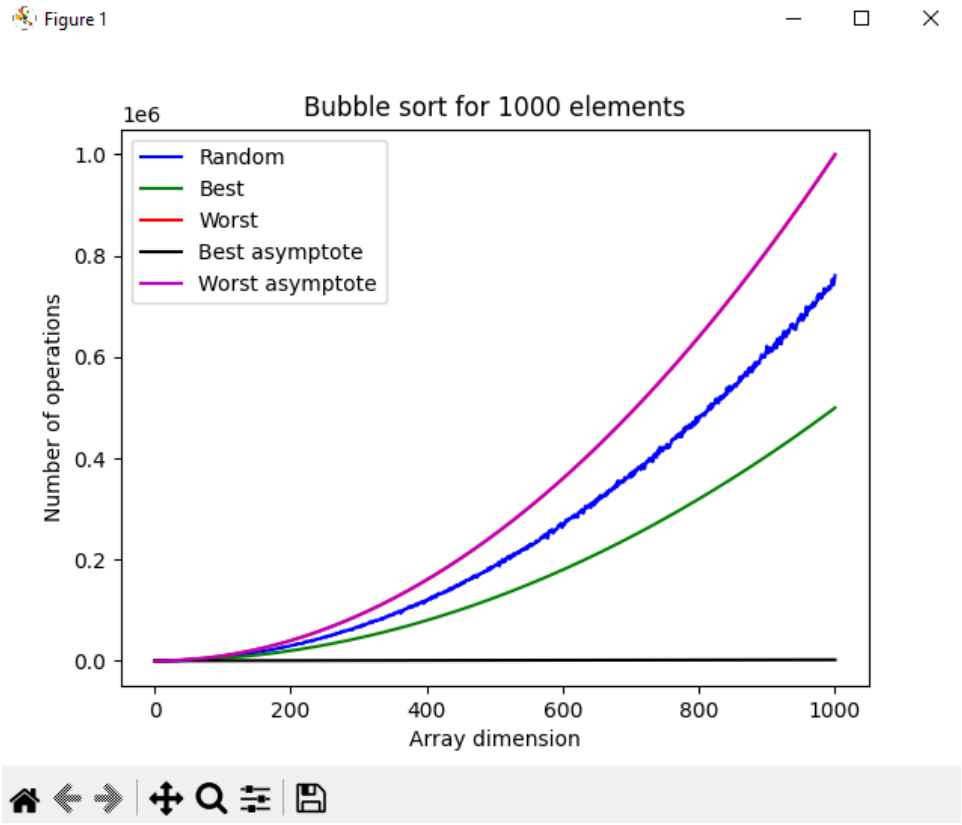
У таблиці 3.4 наведені характеристики оцінювання числа порівнянь і числа перестановок алгоритму сортування бульбашки, модифікованого алгоритму та вставкою для масивів різної розмірності, масиви містять випадкову послідовність елементів. 48

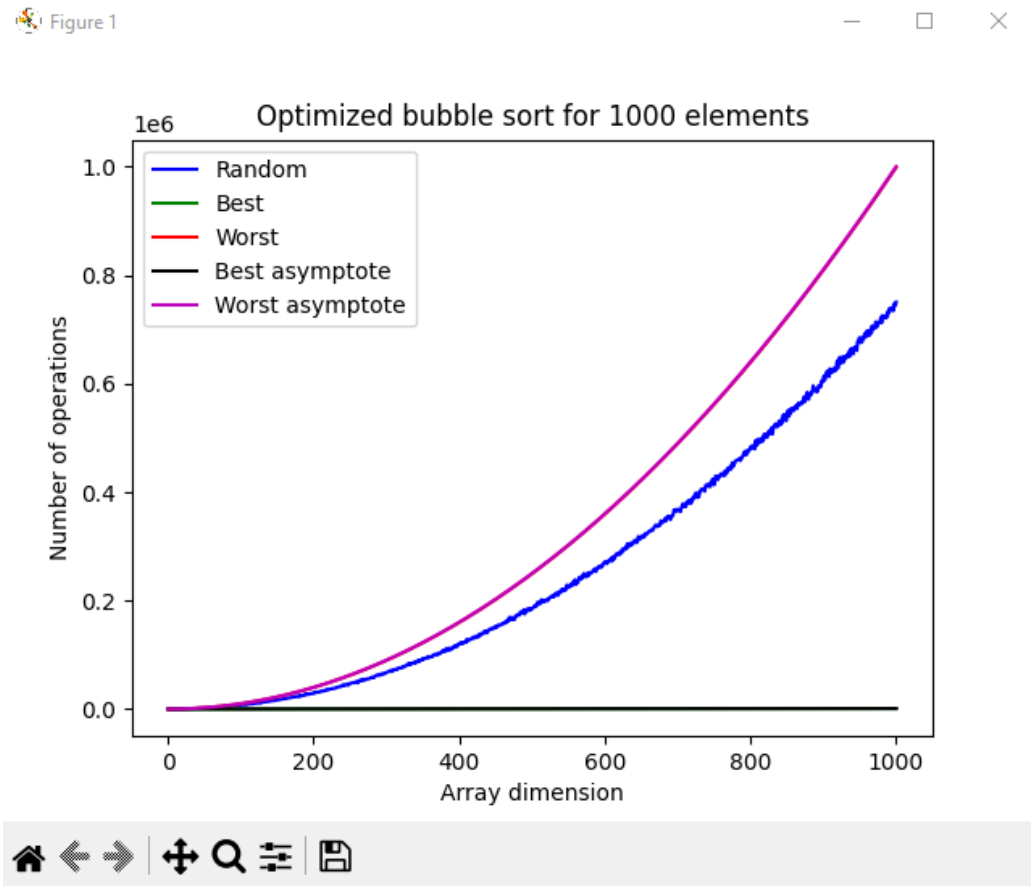
Таблиця 3.4 – Характеристика оцінювання алгоритму сортування бульбашки, модифікованого алгоритму та вставкою для випадкової послідовності елементів у масиві.

Розмірність масиву	Число порівнянь			Число перестановок		
	Звичайне	Модифіковане	Вставка	Звичайне	Модифіковане	Вставка
10	45	44	26	28	22	26
100	4950	4884	2515	2674	2704	2416
1000	499500	498972	257611	267069	242122	256612
5000	12497500	12494574	6256268	6201398	6275052	6251269
10000	49995000	49986872	24974500	25070270	24911680	24964501
20000	199990000	199968472	99388156	99699090	99439488	99368157
50000	1249975000	1249974139	629630015	623546511	624653764	629580016

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.





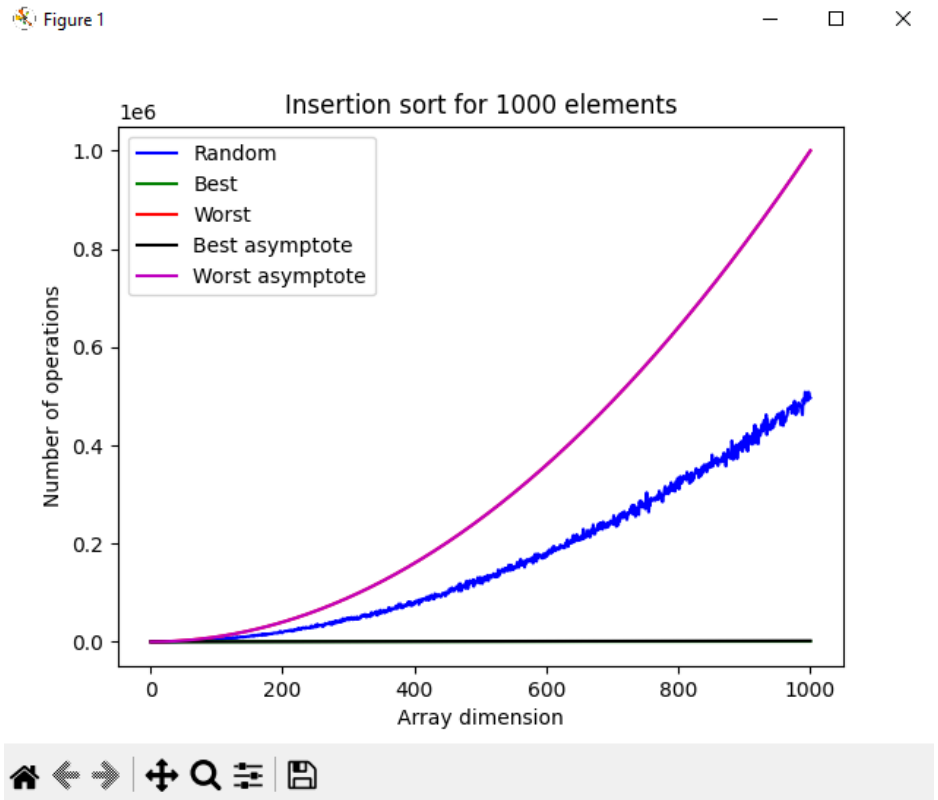


Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

Висновок

Коли я виконував цю лабораторну роботу я вивчив основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування і оцінив їх поріг їх ефективності. Отже, основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування - це вимірювання часу виконання або підрахунок кількості операцій, які виконує алгоритм. Для порівняння різних алгоритмів використовуються асимптотичні нотації, такі як О-нотація, що дозволяють оцінити поведінку алгоритму на різних вхідних даних.

Методи аналізу обчислювальної складності допомагають визначити, наскільки ефективним є алгоритм для великої кількості даних. Внутрішнє сортування є однією з основних операцій при роботі з даними, тому важливо мати знання про ефективність різних алгоритмів сортування.

Застосування правильних методів аналізу допомагає розробникам створювати більш ефективні програми, що можуть працювати з великою кількістю даних. Тому, знання про основні методи аналізу обчислювальної складності алгоритмів внутрішнього сортування є важливим для розробників програмного забезпечення.

КРИТЕРІЇ ОЦІНЮВАННЯ

У випадку здачі лабораторної роботи із захистом максимальний бал дорівнює – 5.

Критерії від максимального балу:

- аналіз алгоритму на відповідність властивостям ;
- псевдокод алгоритму ;
- аналіз часової складності ;
- програмна реалізація алгоритму ;
- тестування алгоритму ;
- висновок;
- захист роботи – 2 бали