**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***

(шифр, прізвище, ім'я, по батькові)

27.02.2023

**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 - О(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 | + | + |

1. Псевдокод алгоритму
   1. Сортування бульбашкою

START

REPEAT

SET swapped to false

FOR i = 1 to index of last element

IF elements at position i and i+1 are out of order

SWAP elements at position i and i+1

SET swapped to true

END IF

END FOR

UNTIL swapped is false

END

* 1. Модифіковане сортування бульбашкою

START

REPEAT

SET swapped to false

SET lastSwap to index of last element

FOR i = 1 to lastSwap

IF elements at position i and i+1 are out of order

SWAP elements at position i and i+1

SET swapped to true

SET lastSwap equal to i

END IF

END FOR

UNTIL swapped is false

END

* 1. Insertion sort

START

FOR i = 1 to index of last element

SET current element equal to elements at position i

WHILE position of current element is greater than 0 AND elements at position of current element - 1 is greater than current element

SWAP elements at position of current element and position of current element - 1

SET current element equal to elements at position of current element - 1

END WHILE

END FOR

END

1. Аналіз часової складності
   1. Сортування бульбашкою

Загальна часова складність алгоритму О(n2).

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
  1. Модифіковане сортування бульбашкою

Загальна часова складність алгоритму О(n2). 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
  1. Insertion sort

Загальна часова складність алгоритму O(n2). 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

1. Програмна реалізація алгоритму
   1. Cортування 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

* 1. 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

* 1. 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

* 1. 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)]

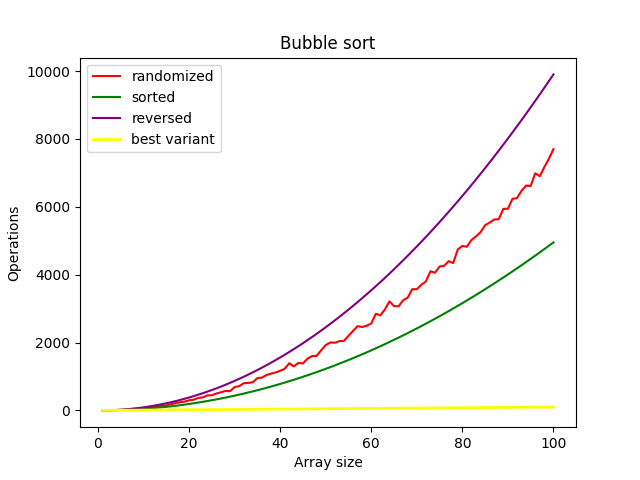
def bubble\_sort(list\_a):  
 number\_of\_comparison = 0  
 number\_of\_permutation = 0  
  
 for i in range(len(list\_a) - 1):  
  
 for j in range(len(list\_a) - i - 1):  
  
 number\_of\_comparison += 1  
 if list\_a[j] > list\_a[j + 1]:  
 number\_of\_permutation += 1  
 list\_a[j], list\_a[j + 1] = 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()

* 1. 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.





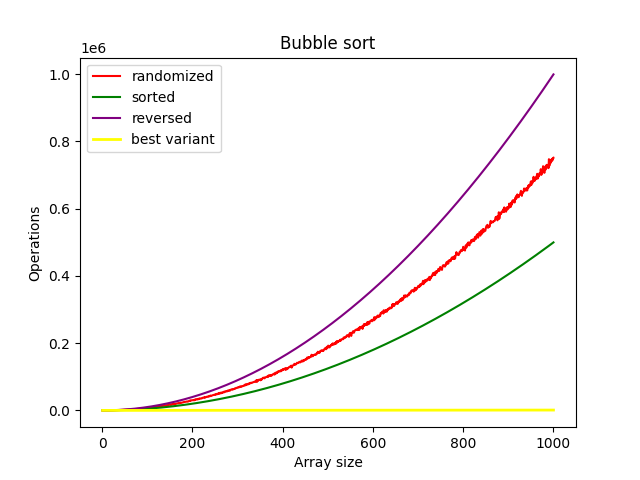


Рисунок 3.1 – Сортування масиву алгоритмом bubble sort

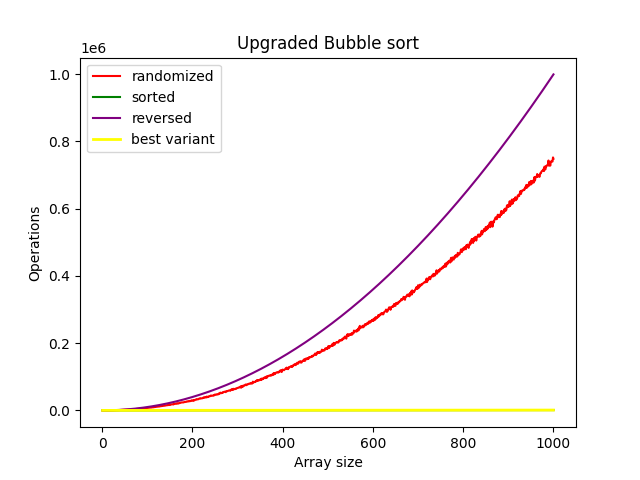
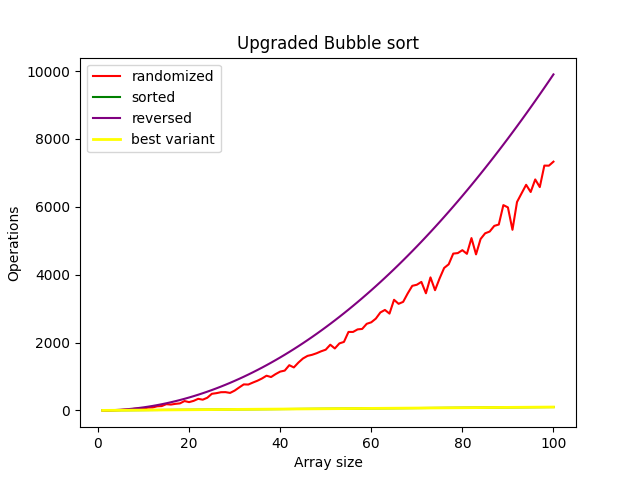
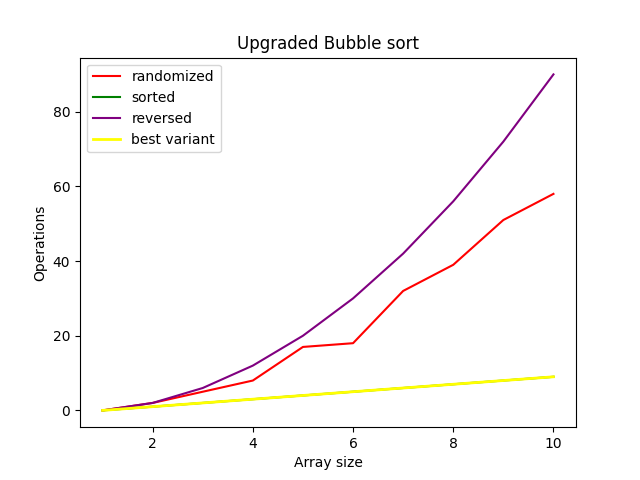


Рисунок 3.2 – Сортування масиву алгоритмом upgraded bubble sort

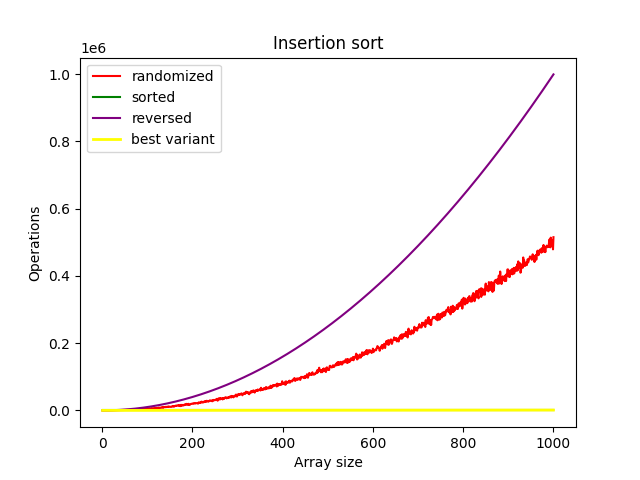
  

Рисунок 3.3 – Сортування масиву алгоритмом insertion sort

**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.

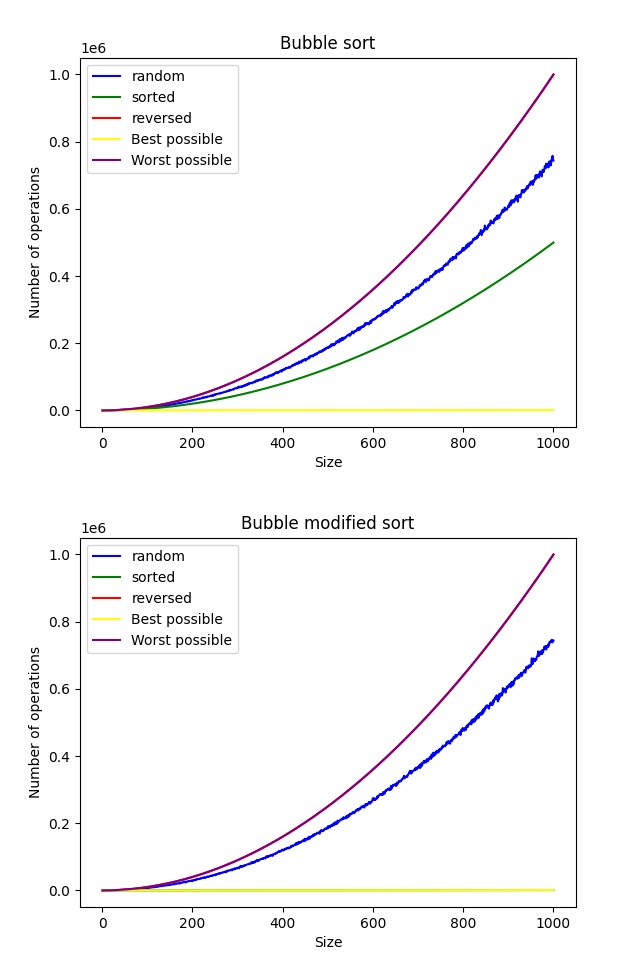
|  |  |  |
| --- | --- | --- |
| Розмірність | Число порівнянь | Число перестановок |
| 10 | 45 45 45 | 45 45 45 |
| 100 | 4950 4950 4950 | 4950 4950 4950 |
| 1000 | 499500 499500 499500 | 499500 499500 499500 |
| 5000 | 12497500 12497500 12497500 | 12497500 12497500 12497500 |
| 10000 | 49995000 49995000 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.

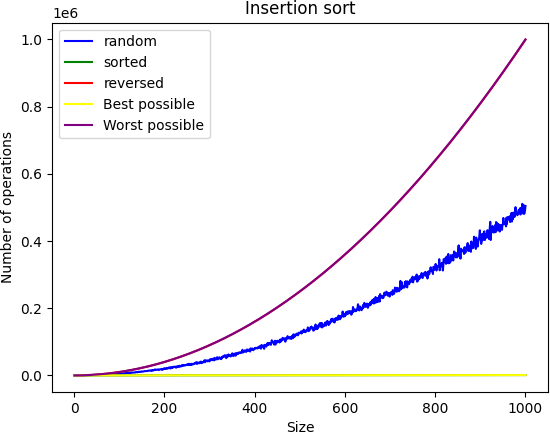


Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

# ВИСНОВОК

На цій лабораторній работі 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(n²). 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.